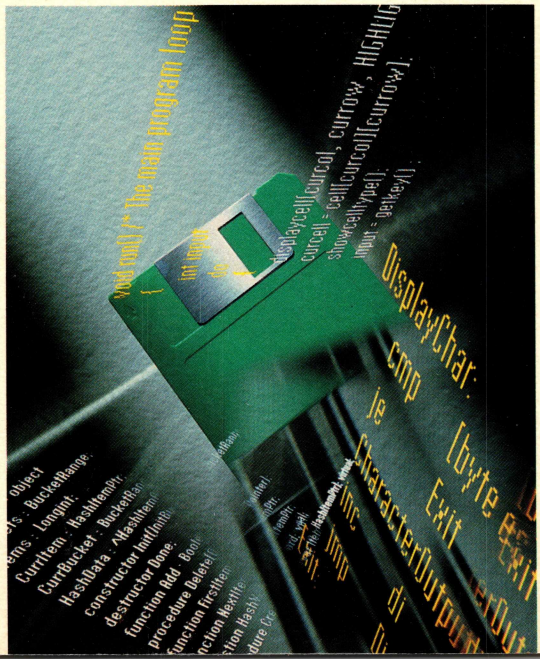
TURBO ASSEMBLER®

2.0

QUICK REFERENCE GUIDE

QUICK REFERENCE GUIDE



BORLAND

Turbo Assembler®

Version 2.0

Quick Reference Guide

BORLAND INTERNATIONAL, INC. 1800 GREEN HILLS ROAD
P.O. BOX 660001, SCOTTS VALLEY, CA 95066-0001

The material in *Part 3* and *Part 4* is reprinted with permission of Intel Corporation, Copyright/Intel Corporation 1987.

Copyright © 1990 by Borland International. All rights reserved. All Borland products are trademarks or registered trademarks of Borland International, Inc. Other brand and product names are trademarks or registered trademarks of their respective holders.

PRINTED IN THE USA.
R1 10 9 8 7 6 5 4 3 2 1

C O N T E N T S

Part 1 Predefined symbols

$ 4
@code 4
@CodeSize 4
@CPU 4
@curseg 4
@data 4
@DataSize 4
??date 4
@fardata 5
@fardata? 5
@FileName 5
??filename 5
@Model 5
@Startup 5
??time 5
??version 5
@WordSize 5

Part 2 Operators

Ideal Mode Operator Precedence 8
MASM Mode Operator Precedence 9
Operators 9
( ) 9
* 9
+ (binary) 9
+ (unary) 10
– (binary) 10
– (unary) 10
. 10
/ 10
: 10
[ ] 11
AND 11
BYTE 11
BYTE PTR 11
CODEPTR 11
DATAPTR 11
DUP 12
DWORD 12
DWORD PTR 12
EQ 12
FAR 12
FAR PTR 12
FWORD 12
FWORD PTR 13
GE 13
GT 13
HIGH 13
HIGH 13
LARGE 13
LE 13
LENGTH 14
LOW 14
LOW 14
LT 14
MASK 14
MOD 14
NE 15
NEAR 15
NEAR PTR 15
NOT 15
OFFSET 15
OR 15
PROC 15
PROC PTR 16
PTR 16
PWORD 16
PWORD PTR 16
QWORD 16
QWORD PTR 16
SEG 16
SHL 17
SHORT 17
SHR 17
SIZE 17

SMALL 17
SYMTYPE 17
TBYTE 18
TBYTE PTR 18
THIS 18
.TYPE 18
TYPE 18
UNKNOWN 18
WIDTH 19
WORD 19
WORD PTR 19
XOR 19
The Special Macro Operators 19
& 19
<> 19
! 20
% 20
;; 20

Part 3 Directives

.186 22
.286 22
.286C 22
.286P 22
.287 22
.386 22
.386C 22
.386P 23
.387 23
.8086 23
.8087 23
: 23
= 23
ALIGN 23
.ALPHA 24
ARG 24
ASSUME 24
%BIN 24
CATSTR 24
.CODE 25
CODESEG 25
COMM 25
COMMENT 25
%CONDS 25
.CONST 26
CONST 26
.CREF 26
%CREF 26
%CREFALL 26
%CREFREF 26
%CREFUREF 26
%CTLS 26
.DATA 27
DATASEG 27
.DATA? 27
DB 27
DD 27
%DEPTH 28
DF 28
DISPLAY 28
DOSSEG 28
DP 28
DQ 29
DT 29
DW 29
ELSE 29
ELSEIF 30
EMUL 30
END 30
ENDIF 30
ENDM 31
ENDP 31
ENDS 31
EQU 31
.ERR 31
ERR 31
.ERR1 31
.ERR2 32
.ERRB 32
.ERRDEF 32
.ERRDIF 32
.ERRDIFI 32
.ERRE 32
.ERRIDN 32
.ERRIDNI 33
ERRIF 33
ERRIF1 33
ERRIF2 33
ERRIFB 33
ERRIFDEF 33
ERRIFDIF 33
ERRIFDIFI 34

ERRIFE 34
ERRIFIDN 34
ERRIFIDNI 34
ERRIFNB 34
ERRIFNDEF 34
.ERRNB 35
.ERRNDEF 35
.ERRNZ 35
EVEN 35
EVENDATA 35
EXITM 35
EXTRN 35
.FARDATA 36
FARDATA 36
.FARDATA? 36
GLOBAL 36
GROUP 37
IDEAL 37
IF 37
IF1 37
IF2 38
IFB 38
IFDEF 38
IFDIF 39
IFDIFI 39
IFE 39
IFIDN 39
IFIDNI 40
IFNB 40
IFNDEF 40
%INCL 41
INCLUDE 41
INCLUDELIB 41
INSTR 41
IRP 41
IRPC 42
JUMPS 42
LABEL 42
.LALL 42
.LFCOND 42
%LINUM 43
%LIST 43
.LIST 43
LOCAL 43
LOCALS 44
MACRO 44
%MACS 44
MASM 44
MASM51 44
.MODEL 44
MODEL 44
MULTERRS 45
NAME 45
%NEWPAGE 45
%NOCONDS 46
%NOCREF 46
%NOCTLS 46
NOEMUL 46
%NOINCL 46
NOJUMPS 46
%NOLIST 46
NOLOCALS 47
%NOMACS 47
NOMASM51 47
NOMULTERRS 47
NOSMART 47
%NOSYMS 47
%NOTRUNC 47
NOWARN 47
ORG 48
%OUT 48
P186 48
P286 48
P286N 48
P286P 48
P287 48
P386 49
P386N 49
P386P 49
P387 49
P8086 49
P8087 49
PAGE 49
%PAGESIZE 49
%PCNT 50
PNO87 50
%POPLCTL 50
PROC 50
PUBLIC 51
PUBLICDLL 51
PURGE 51
%PUSHLCTL 52

QUIRKS 52
.RADIX 52
RADIX 52
RECORD 52
REPT 52
.SALL 53
SEGMENT 53
.SEQ 53
.SFCOND 53
SIZESTR 54
SMART 54
.STACK 54
STACK 54
STRUC 54
SUBSTR 54
SUBTTL 55
%SUBTTL 55
%SYMS 55
%TABSIZE 55
%TEXT 55
.TFCOND 55
TITLE 56
%TITLE 56
%TRUNC 56
UDATASEG 56
UFARDATA 56
UNION 56
USES 57
WARN 57
.XALL 57
.XCREF 57
.XLIST 57

Part 4 Processor Instructions

Operand-size and address-size attributes 60
Default segment attribute .. 60
Operand-size and address-size instruction prefixes 60
Address-size attribute for stack 61
Instruction format 61
ModR/M and SIB bytes ... 63
How to read the instruction set pages 66
Instruction name 67
Flags 67
Opcode 68
/digit 68
/r 68
cb, cw, cd, cp 68
ib, iw, id 68
+rb, +rw, +rd 68
Instruction 69
rel8 69
rel16, rel32 69
ptr16:16, ptr16:32 69
r8 69
r16 69
r32 (80386) 69
imm8 70
imm16 70
imm32 (80386) 70
r/m8 70
r/m16 70
r/m32 70
m8 70
m16 71
m32 71
m16:16, m16:32 (80386) . 71
m16 & 32, m16 & 16 (186/286/386), m32 & 32 (80386) 71
moffs8, moffs16, moffs32 (memory offset; 386 only) 71
Sreg 71
Clocks 71
AAA 72
AAD 73
AAM 74
AAS 74
ADC 75
ADD 76
AND 77
ARPL 78
BOUND 78
BSF 79
BSR 80
BT 80
BTC 81
BTR 81
BTS 82

CALL 82
CBW 84
CDQ 85
CLC 85
CLD 86
CLI 86
CLTS 87
CMC 87
CMP 88
CMPS, CMPSB, CMPSW
CMPSD 89
CWD 90
CWDE 91
DAA 91
DAS 92
DEC 92
DIV 93
ENTER 93
HLT 94
IDIV 95
IMUL 95
IN 97
INC 97
INS, INSB, INSW, INSD 98
INT, INTO 99
IRET, IRETD 101
Jcc 102
JMP 105
LAHF 107
LAR 107
LEA 108
LEAVE 109
LGDT/LIDT 109
LGS, LSS, LFS, LDS, LES 110
LLDT 111
LMSW 112
LOCK 112
LODS, LODSB, LODSW,
LODSD 114
LOOP, LOOPcond 115
LSL 116
LTR 117
MOV 117
MOV 118
MOVS, MOVSB, MOVSW,
MOVSD 119
MOVSX 120
MOVZX 121
MUL 121
NEG 122
NOP 122
NOT 123
OR 123
OUT 124
OUTS, OUTSB, OUTSW,
OUTSD 125
POP 126
POPA, POPAD 127
POPF, POPFD 128
PUSH 129
PUSHA, PUSHAD 130
PUSHF, PUSHFD 130
RCL, RCR, ROL, ROR 131
REP, REPE, REPZ, REPNE,
REPNZ 133
RET 136
SAHF 137
SAL, SAR, SHL, SHR 137
SBB 139
SCAS, SCASB, SCASW,
SCASD 141
SETcc 142
SGDT, SIDT 143
SHLD 144
SHRD 145
SLDT 145
SMSW 146
STC 146
STD 147
STI 147
STOS, STOSB, STOSW,
STOSD 148
STR 149
SUB 149
TEST 150
VERR, VERW 151
WAIT 152
XCHG 153
XLAT, XLATB 153
XOR 154

Part 5 Coprocessor instructions

FABS 159
FADD 159
FADDP 159
FBLD 160
FBSTP 160
FCHS 161
FCLEX, FNCLEX 161
FCOM 161
FCOMP 162
FCOMPP 162
FCOS 162
FDECSTP 163
FDISI, FNDISI 163
FDIV 164
FDIVP 164
FDIVR 164
FDIVRP 165
FENI, FNENI 165
FFREE 166
FIADD 166
FICOM 166
FICOMP 167
FIDIV 167
FIDIVR 168
FILD 168
FIMUL 168
FINCSTP 169
FINIT, FNINIT 169
FIST 170
FISTP 170
FISUB 170
FISUBR 171
FLD 171
FLDCW 172
FLDENV 172
FLDLG2 172
FLDLN2 173
FLDL2E 173
FLDL2T 173
FLDPI 174
FLDZ 174
FLD1 174
FMUL 175
FMULP 175
FNOP 176
FPATAN 176
FPREM 176
FPTAN 177
FRNDINT 177
FRSTOR 177
FSAVE, FNSAVE 178
FSCALE 178
FSETPM 179
FSIN 179
FSINCOS 179
FSQRT 180
FST destination 180
FSTCW, FNSTCW 181
FSTENV, FNSTENV 181
FSTP 181
FSTSW, FNSTSW 182
FSTSW AX, FNSTSW AX 182
FSUB 183
FSUBP 183
FSUBR 183
FSUBRP 184
FTST 184
FUCOM 185
FUCOMP 185
FUCOMPP 185
FWAIT 186
FXAM 186
FXCH 186
FXTRACT 187
FYL2X 187
FYL2XP1 187
F2XM1 188

The *Turbo Assembler Quick-Reference Guide* contains abbreviated discussions of the TASM predefined symbols, operators, and directives in *Parts 1, 2,* and *3,* and a thorough discussion of the processor and coprocessor instructions in *Parts 4* and *5*.

For more complete details about predefined symbols, operators, and directives, see Chapters 1, 2, and 3 in the *Turbo Assembler Reference Guide.*

Several notational conventions are followed in this manual:

- *Italics*: In text, italics represent labels, placeholders, variables, and arrays. In syntax expressions, placeholders are set in italics to indicate that they are user-defined.
- **Boldface**: Boldface is used in text for directives, instructions, symbols, and operators, as well as for command-line options.
- CAPITALS: In text, capital letters are used to represent instructions, directives, registers, and operators.
- `Monospace`: Monospace type is used to display any sample code, text or code that appears on your screen, and any text that you must actually type to assemble, link, and run a program.
- *Keycaps*: In text, keycaps are used to indicate a key on your keyboard. It is often used when describing a key you must press to perform a particular function; for example, "Press *Enter* after typing your program name at the prompt."

P A R T 1

Predefined symbols

All the predefined symbols can be used in both MASM and Ideal mode.

$

Represents the current location counter within the current segment

@code

Alias equate for .CODE segment name

@CodeSize

Numeric equate that indicates code memory model (0=near, 1=far)

@CPU

Numeric equate that returns information about current processor directive

@curseg

Alias equate for current segment

@data

Alias equate for near data group name

@DataSize

Numeric equate that indicates the data memory model (0=near, 1=far, 2=huge)

??date

String equate for today's date

@fardata

Alias equate for initialized far data segment name

@fardata?

Alias equate for uninitialized far data segment name

@FileName

Alias equate for current assembly filename

??filename

String equate for current assembly filename

@Model

Numeric equate representing the model currently in effect.

@Startup

Label that marks the beginning of startup code.

??time

String equate for the current time

??version

Numeric equate for current Turbo Assembler version number

@WordSize

Numeric equate that indicates 16- or 32-bit segments (2=16-bit, 4=32-bit)

P A R T **2**

Operators

This chapter covers the operators Turbo Assembler provides and their precedence. The two tables that follow detail operator precedence for Ideal and MASM modes.

Ideal Mode Operator Precedence

The following table lists the operators in order of priority (highest is first, lowest is last).

- **(), [], LENGTH, MASK, OFFSET, SEG, SIZE, WIDTH**
- **HIGH, LOW**
- **+, –** (unary)
- ***, /, MOD, SHL, SHR**
- **+, –** (binary)
- **EQ, GE, GT, LE, LT, NE**
- **NOT**
- **AND**
- **OR, XOR**
- **:** (segment override)
- **.** (structure member selector)
- **HIGH** (before pointer), **LARGE, LOW** (before pointer), **PTR, SHORT, SMALL, SYMTYPE**

MASM Mode Operator Precedence

- **<>, (), [], LENGTH, MASK, SIZE, WIDTH**
- . (structure member selector)
- **HIGH, LOW**
- **+, –** (unary)
- : (segment override)
- **OFFSET, PTR, SEG, THIS, TYPE**
- ***, /, MOD, SHL, SHR**
- **+, –** (binary)
- **EQ, GE, GT, LE, LT, NE**
- **NOT**
- **AND**
- **OR, XOR**
- **LARGE, SHORT, SMALL, .TYPE**

Operators

( )

Ideal, MASM

(*expression*)

Marks *expression* for priority evaluation.

*

Ideal, MASM

expression1 * *expression2*

Multiplies two integer expressions. Also used with 80386 addressing modes where one expression is a register.

+ (binary)

Ideal, MASM

expression1 + *expression2*

Adds two expressions.

+ (unary) — Ideal, MASM

+ expression

Indicates that *expression* is positive.

– (binary) — Ideal, MASM

expression1 – expression2

Subtracts two expressions.

– (unary) — Ideal, MASM

– expression

Changes the sign of *expression*.

. — Ideal, MASM

memptr.fieldname

Selects a structure member.

/ — Ideal, MASM

expression1 / expression2

Divides two integer expressions.

: — Ideal, MASM

segorgroup : expression

Generates segment or group override.

? — Ideal, MASM

D*x* ?

Initializes with indeterminate data (where D*x* is **DB, DD, DF, DP, DQ, DT,** or **DW**).

[ ] Ideal, MASM

expression1[*expression2*]

[*expression1*][*expression2*]

MASM mode: The **[]** operator can be used to specify addition or register indirect memory operands.

Ideal mode: The **[]** operator specifies a memory reference.

AND Ideal, MASM

expression1 AND *expression2*

Performs a bit-by-bit logical AND of two expressions.

BYTE Ideal

BYTE *expression*

Forces address expression to be byte size.

BYTE PTR Ideal, MASM

BYTE PTR *expression*

Forces address expression to be byte size.

CODEPTR Ideal, MASM

CODEPTR *expression*

Returns the default procedure address size

DATAPTR Ideal

DATAPTR *expression*

Forces address expression to model-dependent size.

DUP — Ideal, MASM

count DUP (*expression* [,*expression*]...)

Repeats a data allocation operation *count* times.

DWORD — Ideal

DWORD *expression*

Forces address expression to be doubleword size.

DWORD PTR — Ideal, MASM

DWORD PTR *expression*

Forces address expression to be doubleword size.

EQ — Ideal, MASM

expression1 EQ *expression2*

Returns true if expressions are equal.

FAR — Ideal

FAR *expression*

Forces an address expression to be a far code pointer.

FAR PTR — Ideal, MASM

FAR PTR *expression*

Forces an address expression to be a far code pointer.

FWORD — Ideal

FWORD *expression*

Forces address expression to be 32-bit far pointer size.

FWORD PTR — Ideal, MASM

FWORD PTR *expression*

Forces address expression to be 32-bit far pointer size.

GE — Ideal, MASM

expression1 GE *expression2*

Returns true if one expression is greater than or equal to the other.

GT — Ideal, MASM

expression1 GT *expression2*

Returns true if one expression is greater than the other.

HIGH — Ideal, MASM

HIGH *expression*

Returns the high part (8 bits or *type* size) of *expression*.

HIGH — Ideal

type HIGH *expression*

Returns the high part (8 bits or *type* size) of *expression*.

LARGE — Ideal, MASM

LARGE *expression*

Sets *expression*'s offset size to 32 bits. In Ideal mode, this operation is legal only if 80386 code generation is enabled.

LE — Ideal, MASM

expression1 LE *expression2*

Returns true if one expression is less than or equal to the other.

LENGTH — Ideal, MASM

LENGTH *name*

Returns number of data elements allocated as part of *name*.

LOW — Ideal, MASM

LOW *expression*

Returns the low part (8 bits or *type* size) of *expression*.

LOW — Ideal

type LOW *expression*

Returns the low part (8 bits or *type* size) of *expression*.

LT — Ideal, MASM

expression1 LT *expression2*

Returns true if one expression is less than the other.

MASK — Ideal, MASM

MASK *recordfieldname*
MASK *record*

Returns a bit mask for a record field or an entire record.

MOD — Ideal, MASM

expression1 MOD *expression2*

Returns remainder (modulus) from dividing two expressions.

NE — Ideal, MASM

expression1 NE *expression2*

Returns true if expressions are not equal.

NEAR — Ideal

NEAR *expression*

Forces an address expression to be a near code pointer.

NEAR PTR — Ideal, MASM

NEAR PTR *expression*

Forces an address expression to be a near code pointer.

NOT — Ideal, MASM

NOT *expression*

Performs a bit-by-bit complement (invert) of *expression*.

OFFSET — Ideal, MASM

OFFSET *expression*

Returns the offset of *expression* within the current segment (or the group that the segment belongs to, if using simplified segmentation directives or Ideal mode).

OR — Ideal, MASM

expression1 OR *expression2*

Performs a bit-by-bit logical OR of two expressions.

PROC — Ideal

PROC *expression*

Forces an address expression to be a near or far code pointer.

PROC PTR — Ideal, MASM

PROC PTR *expression*

Forces an address expression to be a near or far code pointer.

PTR — Ideal, MASM

type PTR *expression*

Forces address expression to have *type* size.

PWORD — Ideal

PWORD *expression*

Forces address expression to be 32-bit far pointer size.

PWORD PTR — Ideal, MASM

PWORD PTR *expression*

Forces address expression to be 32-bit far pointer size.

QWORD — Ideal

QWORD *expression*

Forces address expression to be quadword size.

QWORD PTR — Ideal, MASM

QWORD PTR *expression*

Forces address expression to be quadword size.

SEG — Ideal, MASM

SEG *expression*

Returns the segment address of an expression that references memory.

SHL — Ideal, MASM

expression SHL *count*

Shifts the value of *expression* to the left *count* bits. A negative count causes the data to be shifted the opposite way.

SHORT — Ideal, MASM

SHORT *expression*

Forces *expression* to be a short code pointer (within –128 to +127 bytes of the current code location).

SHR — Ideal, MASM

expression SHR *count*

Shifts the value of *expression* to the right *count* bits. A negative count causes the data to be shifted the opposite way.

SIZE — Ideal, MASM

SIZE *name*

Returns size of data item allocated with *name*. In MASM mode, **SIZE** returns the value of **LENGTH** *name* multiplied by **TYPE** *name*. In Ideal mode, **SIZE** returns the byte count within *name*'s **DUP**.

SMALL — Ideal, MASM

SMALL *expression*

Sets *expression*'s offset size to 16 bits. In Ideal mode, this operation is legal only if 80386 code generation is enabled.

SYMTYPE — Ideal

SYMTYPE <*expression*>

Returns a byte describing *expression*.

TBYTE — Ideal

TBYTE *expression*

Forces address expression to be 10-byte size.

TBYTE PTR — Ideal, MASM

TBYTE PTR *expression*

Forces address expression to be 10-byte size.

THIS — Ideal, MASM

THIS *type*

Creates an operand whose address is the current segment and location counter. *type* describes the size of the operand and whether it refers to code or data.

.TYPE — MASM

.TYPE *expression*

Returns a byte describing the mode and scope of *expression*.

TYPE — Ideal, MASM

TYPE *expression*

Returns a number indicating the size or type of *expression*.

UNKNOWN — Ideal

UNKNOWN *expression*

Removes type information from address expression.

WIDTH — Ideal, MASM

WIDTH *recordfieldname*

WIDTH *record*

Returns the width in bits of a field in a record, or of an entire record.

WORD — Ideal

WORD *expression*

Forces address expression to be word size.

WORD PTR — Ideal, MASM

WORD PTR *expression*

Forces address expression to be word size.

XOR — Ideal, MASM

expression1 XOR *expression2*

Performs bit-by-bit logical exclusive OR of two expressions.

The Special Macro Operators

& — Ideal, MASM

&name

Substitutes actual value of macro parameter *name*.

< > — Ideal, MASM

<text>

Treats *text* literally, regardless of any special characters it might contain.

! Ideal, MASM

!*character*

Treats *character* literally, regardless of any special meaning it might otherwise have.

% Ideal, MASM

%*text*

Treats *text* as an expression, computes its value and replaces *text* with the result. *text* may be either a numeric expression or a text equate.

;; Ideal, MASM

;;*comment*

Suppresses storage of a comment in a macro definition.

P A R T 3

Directives

.186 MASM

Enables assembly of 80186 processor instructions.

.286 MASM

Enables assembly of non-privileged (real mode) 80286 processor instructions and 80287 numeric coprocessor instructions.

.286C MASM

Enables assembly of non-privileged (real mode) 80286 processor instructions and 80287 numeric coprocessor instructions.

.286P MASM

Enables assembly of all 80286 (including privileged mode) processor instructions and 80287 numeric coprocessor instructions.

.287 MASM

Enables assembly of 80287 numeric coprocessor instructions.

.386 MASM

Enables assembly of non-privileged (real mode) 80386 processor instructions and 80387 numeric coprocessor instructions.

.386C MASM

Enables assembly of non-privileged (real mode) 80386 processor instructions and 80387 numeric coprocessor instructions.

.386P — MASM

Enables assembly of all 80386 (including privileged mode) processor instructions and 80387 numeric coprocessor instructions.

.387 — MASM

Enables assembly of 80387 numeric coprocessor instructions.

.8086 — MASM

Enables assembly of 8086 processor instructions only. This is the default processor instruction mode used by Turbo Assembler.

.8087 — MASM

Enables assembly of 8087 numeric coprocessor instructions only. This is the default coprocessor instruction mode used by Turbo Assembler.

: — Ideal, MASM

name:

Defines a near code label called *name*.

= — Ideal, MASM

name = *expression*

Defines or redefines a numeric equate.

ALIGN — Ideal, MASM

ALIGN *boundary*

Rounds up the location counter to a power-of-two address boundary (2, 4, 8, ...).

.ALPHA — MASM

Sets alphanumeric segment-ordering. The **/a** command-line option performs the same function.

ARG — Ideal, MASM

ARG *argument* [,*argument*] ... [=*symbol*]
[RETURNS *argument* [,*argument*]]

Sets up arguments on the stack for procedures. Each argument is assigned a positive offset from the BP register, presuming that both the return address of the procedure call and the caller's BP have been pushed onto the stack already. Each *argument* has the following syntax (boldface items are literal):

```
argname[ [ count1 ] ] [:[distance] PTR] type] [:count2]
```

ASSUME — Ideal, MASM

ASSUME *segmentreg*:*name* [,*segmentreg*:*name*]...
ASSUME *segmentreg*:NOTHING
ASSUME NOTHING

Specifies the segment register (*segmentreg*) that will be used to calculate the effective addresses for all labels and variables defined under a given segment or group name (*name*). The **NOTHING** keyword cancels the association between the designated segment register and segment or group name. The **ASSUME NOTHING** statement removes all associations between segment registers and segment or group names.

%BIN — Ideal, MASM

%BIN *size*

Sets the width of the object code field in the listing file to *size* columns.

CATSTR — Ideal, MASM51

name CATSTR *string* [,*string*]...

Concatenates several strings to form a single string *name*.

.CODE — MASM

CODESEG — Ideal, MASM

.CODE [*name*]
CODESEG [*name*]

Defines the start of a code segment when used with the **.MODEL** directive. If you have specified the medium or large memory model, you can follow the **.CODE** (or **CODESEG**) directive with an optional name that indicates the name of the segment.

COMM — Ideal, MASM

COMM *definition* [*,definition*]...

Defines a communal variable. Each definition describes a symbol and has the following format:

```
[distance] symbolname:type [:count]
```

distance can be either **NEAR** or **FAR** and defaults to the size of the default data memory model if not specified. *symbolname* is the symbol that is to be communal. If *distance* is **FAR**, *symbolname* may also specify an array element size multiplier to be included in the total space computation (*name*[*multiplier*]). *type* is one of: **BYTE, WORD, DATAPTR, CODEPTR, DWORD, FWORD, PWORD, QWORD, TBYTE,** or a structure name. *count* specifies how many items this communal symbol defines (default is 1).

COMMENT — MASM

COMMENT *delimiter* [*text*]
[*text*]
delimiter [*text*]

Starts a multiline comment. *delimiter* is the first non-blank character following **COMMENT**.

%CONDS — Ideal, MASM

Shows all statements in conditional blocks in the listing. This is the default mode for Turbo Assembler.

.CONST — MASM

CONST — Ideal, MASM

Defines the start of the constant data segment.

.CREF — MASM

%CREF — Ideal, MASM

Allows cross-reference information to be accumulated for all symbols encountered from this point forward in the source file. **.CREF** reverses the effect of any **%XCREF** or **.XCREF** directives that inhibited the information collection.

%CREFALL — Ideal, MASM

Causes all subsequent symbols in the source file to appear in the cross-reference listing. This is the default mode for Turbo Assembler. **%CREFALL** reverses the effect of any previous **%CREFREF** or **%CREFUREF** directives that disabled the listing of unreferenced or referenced symbols.

%CREFREF — Ideal, MASM

Disables listing of unreferenced symbols in cross-reference.

%CREFUREF — Ideal, MASM

Lists only the unreferenced symbols in cross-reference.

%CTLS — Ideal, MASM

Causes listing control directives (such as **%LIST**, **%INCL**, and so on) to be placed in the listing file.

.DATA — MASM

DATASEG — Ideal

Defines the start of the initialized data segment in your module. You must first have used the **.MODEL** directive to specify a memory model. The data segment is put in a group called DGROUP, which also contains the segments defined with the **.STACK**, **.CONST**, and **.DATA?** directives.

.DATA? — MASM

Defines the start of the uninitialized data segment in your module. You must first have used the **.MODEL** directive to specify a memory model. The data segment is put in a group called DGROUP, which also contains the segments defined with the **.STACK**, **.CONST**, and **.DATA** directives.

DB — Ideal, MASM

[*name*] DB *expression* [,*expression*]...

Allocates and initializes a byte of storage. *name* is the symbol you'll subsequently use to refer to the data. *expression* can be a constant expression, a question mark, a character string, or a **DUP**licated expression.

DD — Ideal, MASM

[*name*] DD [*type* PTR] *expression* [,*expression*]...

Allocates and initializes 4 bytes (a doubleword) of storage. *name* is the symbol you'll subsequently use to refer to the data. *type* followed by **PTR** adds debug information to the symbol being defined, so that Turbo Debugger can display its contents properly. *type* is one of the following: **BYTE, WORD, DATAPTR, CODEPTR, DWORD, FWORD, PWORD, QWORD, TBYTE, SHORT, NEAR, FAR** or a structure name. *expression* can be a constant expression, a 32-bit floating-point number, a question mark, an address expression, or a **DUP**licated expression.

%DEPTH — Ideal, MASM

%DEPTH *width*

Sets size of depth field in listing file to *width* columns. The default is 1 column.

DF — Ideal, MASM

[*name*] DF [*type* PTR] *expression* [,*expression*]...

Allocates and initializes 6 bytes (a far 48-bit pointer) of storage. name is the symbol you'll subsequently use to refer to the data. *type* followed by **PTR** adds debug information to the symbol being defined, so that Turbo Debugger can display its contents properly. *type* is one of the following: **BYTE, WORD, DATAPTR, CODEPTR, DWORD, FWORD, PWORD, QWORD, TBYTE, SHORT, NEAR, FAR** or a structure name. *expression* can be a constant expression, a question mark, an address expression, or a **DUP**licated expression.

DISPLAY — Ideal, MASM

DISPLAY *"text"*

Outputs a quoted string (*text*) to the screen.

DOSSEG — Ideal, MASM

Enables DOS segment-ordering at link time.

DP — Ideal, MASM

[*name*] **DP** [*type* PTR] *expression* [,*expression*]...

Allocates and initializes 6 bytes (a far 48-bit pointer) of storage. name is the symbol you'll subsequently use to refer to the data. *type* followed by **PTR** adds debug information to the symbol being defined, so that Turbo Debugger can display its contents properly. *type* is one of the following: **BYTE, WORD, DATAPTR, CODEPTR, DWORD, FWORD, PWORD, QWORD, TBYTE, SHORT, NEAR, FAR** or a structure name. *expression* can be a constant expression, a question mark, an address expression, or a **DUP**licated expression.

DQ — Ideal, MASM

[*name*] DQ *expression* [,*expression*]...

Allocates and initializes 8 bytes (a quadword) of storage. *name* is the symbol you'll subsequently use to refer to the data. expression can be a constant expression, a 64-bit floating-point number, a question mark, or a **DUP**licated expression.

DT — Ideal, MASM

[*name*] DT *expression* [,*expression*]...

Allocates and initializes 10 bytes of storage. name is the symbol you'll subsequently use to refer to the data. *expression* can be a constant expression, a packed decimal constant expression, a question mark, an 80-bit floating-point number, or a **DUP**licated expression.

DW — Ideal, MASM

[*name*] DW [*type* PTR] *expression* [,*expression*]...

Allocates and initializes 2 bytes (a word) of storage. *name* is the symbol you'll subsequently use to refer to the data. *type* followed by **PTR** adds debug information to the symbol being defined, so that Turbo Debugger can display its contents properly. *type* is one of the following: **BYTE, WORD, DATAPTR, CODEPTR, DWORD, FWORD, PWORD, QWORD, TBYTE, SHORT, NEAR, FAR** or a structure name. expression can be a constant expression, a question mark, an address expression, or a **DUP**licated expression.

ELSE — Ideal, MASM

ELSE

IF *condition*
  statements1
[ELSE
  statements2]
ENDIF

Starts alternative conditional assembly block. The statements introduced by ELSE (*statements2*) are assembled if *condition* evaluates to false.

ELSEIF — Ideal, MASM

ELSEIF

IF *condition1*
  statements1
[ELSEIF *condition2*
  statements2]
ENDIF

Starts nested conditional assembly block if *condition2* is true. Several other forms of **ELSEIF** are supported: **ELSEIF1, ELSEIF2, ELSEIFB, ELSEIFDEF, ELSEIFDIF, ELSEIFDIFI, ELSEIFE, ELSEIFIDN, ELSEIFIDNI, ELSEIFNB,** and **ELSEIFNDEF.**

EMUL — Ideal, MASM

Causes all subsequent numeric coprocessor instructions to be generated as emulated instructions, instead of real instructions. When your program is executed, you must have a software floating-point emulation package installed or these instructions will not work properly.

END — Ideal, MASM

END [*startaddress*]

Marks the end of a source file. *startaddress* is a symbol or expression that specifies the address in your program where you want execution to begin. Turbo Assembler ignores any text that appears after the **END** directive.

ENDIF — Ideal, MASM

ENDIF

IF *condition*
  statements
ENDIF

Marks the end of a conditional assembly block started with one if the **IF***xxxx* directives.

ENDM — Ideal, MASM

Marks the end of a repeat block or a macro definition.

ENDP — Ideal, MASM

ENDP [*procname*]
[*procname*] ENDP

Marks the end of a procedure. If *procname* is supplied, it must match the procedure name specified with the **PROC** directive that started the procedure definition.

ENDS — Ideal, MASM

ENDS [*segmentname* | *strucname*]
[*segmentname* | *strucname*]ENDS

Marks end of current segment, structure or union. If you supply the optional name, it must match the name specified with the corresponding **SEGMENT**, **STRUC**, or **UNION** directive.

EQU — Ideal, MASM

name EQU *expression*

Defines *name* to be a string, alias, or numeric equate containing the result of evaluating *expression*.

.ERR — MASM

ERR — Ideal, MASM

Forces an error to occur at the line that this directive is encountered on in the source file.

.ERR1 — MASM

Forces an error to occur on pass 1 of assembly.

.ERR2 MASM

Forces an error to occur on pass 2 of assembly if multiple-pass mode (controlled by **/m** command-line option) is enabled.

.ERRB MASM

.ERRB *<argument>*

Forces an error to occur if *argument* is blank (empty).

.ERRDEF MASM

.ERRDEF *symbol*

Forces an error to occur if *symbol* is defined.

.ERRDIF MASM

.ERRDIF *<argument1>,<argument2>*

Forces an error to occur if arguments are different. The comparison is case-sensitive.

.ERRDIFI MASM

.ERRDIFI *<argument1>,<argument2>*

Forces an error to occur if arguments are different. The comparison is not case-sensitive.

.ERRE MASM

.ERRE *expression*

Forces an error to occur if *expression* is false (0).

.ERRIDN MASM

.ERRIDN *<argument1>,<argument2>*

Forces an error to occur if arguments are identical. The comparison is case-sensitive.

.ERRIDNI — MASM

.ERRIDNI <*argument1*>,<*argument2*>

Forces an error to occur if arguments are identical. The comparison is not case-sensitive.

ERRIF — Ideal, MASM

ERRIF *expression*

Forces an error to occur if *expression* is true (nonzero).

ERRIF1 — Ideal, MASM

Forces an error to occur on pass 1 of assembly.

ERRIF2 — Ideal, MASM

Forces an error to occur on pass 2 of assembly if multiple-pass mode (controlled by **/m** command-line option) is enabled.

ERRIFB — Ideal, MASM

ERRIFB <*argument*>

Forces an error to occur if *argument* is blank (empty).

ERRIFDEF — Ideal, MASM

ERRIFDEF *symbol*

Forces an error if *symbol* is defined.

ERRIFDIF — Ideal, MASM

ERRIFDIF <*argument1*>,<*argument2*>

Forces an error to occur if arguments are different. The comparison is case-sensitive.

ERRIFDIFI — Ideal, MASM

ERRIFDIFI *<argument1>,<argument2>*

Forces an error to occur if arguments are different. The comparison is not case-sensitive.

ERRIFE — Ideal, MASM

ERRIFE *expression*

Forces an error if *expression* is false (0).

ERRIFIDN — Ideal, MASM

ERRIFIDN *<argument1>,<argument2>*

Forces an error to occur if arguments are identical. The comparison is case-sensitive.

ERRIFIDNI — Ideal, MASM

ERRIFIDNI *<argument1>,<argument2>*

Forces an error to occur if arguments are identical. The comparison is not case-sensitive.

ERRIFNB — Ideal, MASM

ERRIFNB *<argument>*

Forces an error to occur if *argument* is not blank.

ERRIFNDEF — Ideal, MASM

ERRIFNDEF *symbol*

Forces an error to occur if *symbol* is not defined.

.ERRNB — MASM

.ERRNB *<argument>*

Forces an error to occur if *argument* is not blank.

.ERRNDEF — MASM

.ERRNDEF *symbol*

Forces an error to occur if *symbol* is not defined.

.ERRNZ — MASM

.ERRNZ *expression*

Forces an error to occur if *expression* is true (nonzero).

EVEN — Ideal, MASM

Rounds up the location counter to the next even address.

EVENDATA — Ideal, MASM

Rounds up the location counter to the next even address in a data segment.

EXITM — Ideal, MASM

Terminates macro- or block-repeat expansion and returns control to the next statement following the macro or repeat-block call.

EXTRN — Ideal, MASM

EXTRN *definition* [,*definition*]...

Indicates that a symbol is defined in another module. *definition* describes a symbol and has the following format:

```
[language] name:type [:count]
```

language specifies that the naming conventions of **C, PASCAL, BASIC, FORTRAN, ASSEMBLER,** or **PROLOG** are to be applied to symbol *name*. *name* is the symbol that is defined in another module. type must match the type of the symbol where it's defined and must be one of the following: **NEAR, FAR, PROC, BYTE, WORD, DWORD, DATAPTR, CODEPTR, FWORD, PWORD, QWORD, TBYTE, ABS,** or a structure name. *count* specifies how many items this external symbol defines and defaults to one if not specified.

.FARDATA — MASM

FARDATA — Ideal, MASM

.FARDATA [*segmentname*]
FARDATA [*segmentname*]

Defines the start of a far initialized data segment. *segmentname*, if present, overrides the default segment name.

.FARDATA? — MASM

.FARDATA? [*segmentname*]

Defines the start of a far uninitialized data segment. *segmentname*, if present, overrides the default segment name.

GLOBAL — Ideal, MASM

GLOBAL *definition* [,*definition*]...

Acts as a combination of the **EXTRN** and **PUBLIC** directives to define a global symbol. *definition* describes the symbol and has the following format:

```
[language] name:type [:count]
```

language specifies that the naming conventions of **C, PASCAL, BASIC, FORTRAN, ASSEMBLER,** or **PROLOG** are to be applied to symbol *name*. If *name* is defined in the current source file, it is made public exactly as if used in a **PUBLIC** directive. If not, it is declared as an external symbol of type *type*, as if the **EXTRN** directive had been used. *type* must match the type of the symbol in the module where it is defined, and must be one of the following: **NEAR, FAR, PROC, BYTE, WORD, DATAPTR, CODEPTR,**

DWORD, FWORD, PWORD, QWORD, TBYTE, ABS, or a structure name. *count* specifies how many items this symbol defines (one is the default).

GROUP — Ideal, MASM

GROUP *groupname segmentname* [,*segmentname*]...
groupname GROUP *segmentname* [,*segmentname*]...

Associates *groupname* with one or more segments, so that all labels and variables defined in those segments have their offsets computed relative to the beginning of group *groupname*. *segmentname* can be either a segment name defined previously with **SEGMENT** or an expression starting with **SEG**. In MASM mode, you must use a group override whenever you access a symbol in a segment that is part of a group. In Ideal mode, Turbo Assembler automatically generates group overrides for such symbols.

IDEAL — Ideal, MASM

Enters Ideal assembly mode. Ideal mode will stay in effect until it is overridden by a **MASM** or **QUIRKS** directive.

IF — Ideal, MASM

IF

IF *expression*
  truestatements
[ELSE
  falsestatements]
ENDIF

Initiates a conditional block, causing the assembly of *truestatements* up to the optional **ELSE** directive, provided that *expression* is true (nonzero).

IF1 — Ideal, MASM

IF1

IF1
  truestatements
[ELSE
  falsestatements]
ENDIF

Initiates a conditional block, causing the assembly of *truestatements* up to the optional **ELSE** directive, provided that the current assembly pass is pass one.

IF2 — Ideal, MASM

IF2

IF2
  truestatements
[ELSE
  falsestatements]
ENDIF

Initiates a conditional block, causing the assembly of *truestatements* up to the optional **ELSE** directive, provided that multiple-pass mode (controlled by the **/m** command-line option) is enabled and the current assembly pass is pass two.

IFB — Ideal, MASM

IFB

IFB <*argument*>
  truestatements
[ELSE
  falsestatements]
ENDIF

Initiates a conditional block, causing the assembly of *truestatements* up to the optional **ELSE** directive, provided that *argument* is blank (empty).

IFDEF — Ideal, MASM

IFDEF

IFDEF *symbol*
  truestatements
[ELSE
  falsestatements]
ENDIF

Initiates a conditional block, causing the assembly of *truestatements* up to the optional **ELSE** directive, provided that *symbol* is defined.

IFDIF — Ideal, MASM

IFDIF

IFDIF <*argument1*>,<*argument2*>
  truestatements
[ELSE
  falsestatements]
ENDIF

Initiates a conditional block, causing the assembly of *truestatements* up to the optional **ELSE** directive, provided that the arguments are different. The comparison is case-sensitive.

IFDIFI — Ideal, MASM

IFDIFI

IFDIFI <*argument1*>,<*argument2*>
  truestatements
[ELSE
  falsestatements]
ENDIF

Initiates a conditional block, causing the assembly of *truestatements* up to the optional **ELSE** directive, provided that the arguments are different. The comparison is not case-sensitive.

IFE — Ideal, MASM

IFE

IFE *expression*
  truestatements
[ELSE
  falsestatements]
ENDIF

Initiates a conditional block, causing the assembly of *truestatements* up to the optional **ELSE** directive, provided that *expression* is false.

IFIDN — Ideal, MASM

IFIDN

IFIDN *<argument1>,<argument2>*
  truestatements
[ELSE
  falsestatements]
ENDIF

Initiates a conditional block, causing the assembly of *truestatements* up to the optional **ELSE** directive, provided that the arguments are identical. The comparison is case-sensitive.

IFIDNI — Ideal, MASM

IFIDNI

IFIDNI *<argument1>,<argument2>*
  truestatements
[ELSE
  falsestatements]
ENDIF

Initiates a conditional block, causing the assembly of *truestatements* up to the optional **ELSE** directive, provided that the arguments are identical. The comparison is not case-sensitive.

IFNB — Ideal, MASM

IFNB

IFNB *<argument>*
  truestatements
[ELSE
  falsestatements]
ENDIF

Initiates a conditional block, causing the assembly of *truestatements* up to the optional **ELSE** directive, provided that *argument* is nonblank.

IFNDEF — Ideal, MASM

IFNDEF

IFNDEF *symbol*
  truestatements
[ELSE
  falsestatements]
ENDIF

Initiates a conditional block, causing the assembly of *truestatements* up to the optional **ELSE** directive, provided that *symbol* is not defined.

%INCL — Ideal, MASM

Enables listing of include files. This is the default **INCLUDE** file listing mode.

INCLUDE — MASM, Ideal

INCLUDE *filename* or INCLUDE *"filename"*

Includes source code from file *filename* at the current position in the module being assembled. If no extension is specified, .ASM is assumed.

INCLUDELIB — MASM, Ideal

INCLUDELIB *filename* or INCLUDELIB *"filename"*

Causes the linker to include library *filename* at link time. If no extension is specified, .LIB is assumed.

INSTR — Ideal, MASM51

name INSTR [*start*,]*string1*,*string2*

name is assigned the position of the first instance of *string2* in *string1*. Searching begins at position *start* (position one if *start* not specified). If *string2* does not appear anywhere within *string1*, *name* is set to zero.

IRP — Ideal, MASM

IRP *parameter*,<*arg1*[,*arg2*]...>
  statements
ENDM

Repeats a block of statements with string substitution. *statements* are assembled once for each argument present. The arguments may be any text, such as symbols, strings, numbers, and so on. Each time the block is assembled, the next argument in the list is substituted for any instance of *parameter* in the *statements*.

IRPC — Ideal, MASM

IRPC *parameter,string*
  statements
ENDM

Repeats a block of statements with character substitution. *statements* are assembled once for each character in *string*. Each time the block is assembled, the next character in the string is substituted for any instances of *parameter* in *statements*.

JUMPS — Ideal, MASM

Causes Turbo Assembler to look at the destination address of a conditional jump instruction, and if it is too far away to reach with the short displacement that these instructions use, it generates a conditional jump of the opposite sense around an ordinary jump instruction to the desired target address. This directive has the same effect as using the **/JJUMPS** command-line option.

LABEL — MASM, Ideal

name LABEL *type*
LABEL *name type*

Defines a symbol *name* to be of type *type*. *name* must not have been defined previously in the source file. *type* must be one of the following: **NEAR, FAR, PROC, BYTE, WORD, DATAPTR, CODEPTR, DWORD, FWORD, PWORD, QWORD, TBYTE,** or a structure name.

.LALL — MASM

Enables listing of macro expansions.

.LFCOND — MASM

Shows all statements in conditional blocks in the listing.

%LINUM — Ideal, MASM

%LINUM *size*

Sets the width of the line-number field in listing file to *size* columns. The default is four columns.

%LIST — Ideal, MASM

.LIST — MASM

Shows source lines in the listing. This is the default listing mode.

LOCAL — Ideal, MASM

In macros:

```
LOCAL symbol [,symbol]...
```

In procedures:

```
LOCAL localdef [,localdef]... [=symbol]
```

Defines local variables for macros and procedures. Within a macro definition, **LOCAL** defines temporary symbol names that are replaced by new unique symbol names each time the macro is expanded. **LOCAL** must appear before any other statements in the macro definition. Within a procedure, **LOCAL** defines names that access stack locations as negative offsets relative to the BP register. If you end the argument list with an equal sign (**=**) and a symbol, that symbol will be equated to the total size of the local symbol block in bytes. Each *localdef* has the following syntax:

```
localname:[ [distance] PTR]type[:count]
```

or

```
localname[ [count] ][:[distance] PTR]type]
```

localname is the name you'll use to refer to this local symbol throughout the procedure. *type* is the data type of the argument and can be one of the following: **WORD, DATAPTR, CODEPTR, DWORD, FWORD, PWORD, QWORD, TBYTE,** or a structure name. If *type* is not specified using the alternative syntax, **WORD** size is assumed. *count* specifies how many elements of the specified type to allocate on the stack. Using *distance* and

PTR includes debugging information for Turbo Debugger, telling it that this local variable is really a pointer to another data type.

LOCALS — Ideal, MASM

LOCALS [*prefix*]

Enables local symbols, whose names will begin with two at-signs (@@) or the two-character *prefix* if it is specified. Local symbols are automatically enabled in Ideal mode.

MACRO — Ideal, MASM

MACRO *name* [*parameter* [,*parameter*]...]
name MACRO [*parameter* [,*parameter*]...]

Defines a macro to be expanded later when *name* is encountered. *parameter* is a placeholder that you use in the the body of the macro definition wherever you want to substitute one of the actual arguments the macro is called with.

%MACS — Ideal, MASM

Enables listing of macro expansions.

MASM — Ideal, MASM

Enters MASM assembly mode. This is the default assembly mode for Turbo Assembler.

MASM51 — Ideal, MASM

Enables assembly of some MASM 5.1 enhancements.

.MODEL — MASM

MODEL — Ideal, MASM

.MODEL [*model modifier*] *memorymodel* [*modulename*]

[,[*language modifier*] *language*]

.MODEL TPASCAL

MODEL [*model modifier*] *memorymodel* [*modulename*]
[,[*language modifier*] *language*]

MODEL TPASCAL

Sets the memory model for simplified segmentation directives. *model modifier* is **SS_NE_DS** if present. *memorymodel* is **TINY, SMALL, MEDIUM, COMPACT, LARGE,** or **HUGE.** *modulename* is used in the large models to declare the name of the code segment. *language modifier* is either **WINDOWS** or **NOWINDOWS,** to specify generation of MSWindows procedure entry/exit code. *language* specifies which language you will be calling from to access the procedures in this module: **C, PASCAL, BASIC, FORTRAN, ASSEMBLER,** or **PROLOG.** Turbo Assembler automatically generates the appropriate procedure entry and exit code when you use the **PROC/ENDP** directives. *language* also tells Turbo Assembler which naming conventions to use for public and external symbols, and in what order procedure arguments were pushed onto the stack by the calling module. Also, the appropriate form of the **RET** instruction is generated to remove the arguments from the stack before returning if required. The **TPASCAL** form is used when writing an assembler module that interfaces with Turbo Pascal.

MULTERRS — Ideal, MASM

Allows multiple errors to be reported on a single source line.

NAME — Ideal, MASM

NAME *modulename*

Sets the object file's module name. This directive has no effect in MASM mode; it only works in Ideal mode.

%NEWPAGE — Ideal, MASM

Starts a new page in the listing file.

%NOCONDS — Ideal, MASM

Disables the placement of statements in conditional blocks in the listing file.

%NOCREF — Ideal, MASM

%NOCREF [*symbol, ...*]

Disables cross-reference listing (CREF) information accumulation. If you supply one or more symbol names, cross-referencing is disabled only for those symbols.

%NOCTLS — Ideal, MASM

Disables placement of listing-control directives in the listing file. This is the default listing-control mode for Turbo Assembler.

NOEMUL — Ideal, MASM

Causes all subsequent numeric coprocessor instructions to be generated as real instructions, instead of emulated instructions. When your program is executed, you must have an 80x87 coprocessor installed or these instructions will not work properly. This is the default floating-point assembly mode for Turbo Assembler.

%NOINCL — Ideal, MASM

Disables listing of source lines from **INCLUDE** files.

NOJUMPS — Ideal, MASM

Disables stretching of conditional jumps enabled with **JUMPS**. This is the default mode for Turbo Assembler.

%NOLIST — Ideal, MASM

Disables output to the listing file.

NOLOCALS — Ideal, MASM

Disables local symbols enabled with **LOCALS**. This is the default for Turbo Assembler's MASM mode.

%NOMACS — Ideal, MASM

Lists only macro expansions that generate code. This is the default macro listing mode for Turbo Assembler.

NOMASM51 — Ideal, MASM

Disables assembly of certain MASM 5.1 enhancements enabled with **MASM51**. This is the default mode for Turbo Assembler.

NOMULTERRS — Ideal, MASM

Allows only a single error to be reported on a source line. This is the default error-reporting mode for Turbo Assembler.

NOSMART — Ideal, MASM

Disables code optimizations that generate different code than MASM.

%NOSYMS — Ideal, MASM

Disables placement of the symbol table in the listing file.

%NOTRUNC — Ideal, MASM

Prevents truncation of fields whose contents are longer than the corresponding field widths in the listing file.

NOWARN — Ideal, MASM

NOWARN [*warnclass*]

Disables warning messages with warning identifier *warnclass*, or all warning messages if *warnclass* is not specified.

ORG — Ideal, MASM

ORG *expression*

Sets the location counter in the current segment to the address specified by *expression*.

%OUT — MASM

%OUT *text*

Displays *text* on screen.

P186 — Ideal, MASM

Enables assembly of 80186 processor instructions.

P286 — Ideal, MASM

Enables assembly of all 80286 (including privileged mode) processor instructions and 80287 numeric coprocessor instructions.

P286N — Ideal, MASM

Enables assembly of non-privileged (real mode) 80286 processor instructions and 80287 numeric coprocessor instructions.

P286P — Ideal, MASM

Enables assembly of all 80286 (including privileged mode) processor instructions and 80287 numeric coprocessor instructions.

P287 — Ideal, MASM

Enables assembly of 80287 numeric coprocessor instructions.

P386 — Ideal, MASM

Enables assembly of all 80386 (including privileged mode) processor instructions and 80387 numeric coprocessor instructions.

P386N — Ideal, MASM

Enables assembly of non-privileged (real mode) 80386 processor instructions and 80387 numeric coprocessor instructions.

P386P — Ideal, MASM

Enables assembly of all 80386 (including privileged mode) processor instructions and 80387 numeric coprocessor instructions.

P387 — Ideal, MASM

Enables assembly of 80387 numeric coprocessor instructions.

P8086 — Ideal, MASM

Enables assembly of 8086 processor instructions only. This is the default processor instruction mode for Turbo Assembler.

P8087 — Ideal, MASM

Enables assembly of 8087 numeric coprocessor instructions only. This is the default coprocessor instruction mode for Turbo Assembler.

PAGE — MASM

%PAGESIZE — Ideal, MASM

PAGE [*rows*] [,*cols*]
%PAGESIZE [*rows*] [,*cols*]

Sets the listing page height and width, starts new pages. *rows* specifies the number of lines that will appear on each listing page (10..255). *cols* specifies the number of columns wide the page will be (59..255). Omitting *rows* or *cols* leaves the current setting unchanged. If you follow **PAGE** with a plus sign (+), a new page starts, the section number is incremented, and the page number restarts at 1. **PAGE** with no arguments forces the listing to resume on a new page, with no change in section number.

%PCNT — Ideal, MASM

%PCNT *width*

Sets segment:offset field width in listing file to *width* columns. The default is 4 for 16-bit segments and 8 for 32-bit segments.

PNO87 — Ideal, MASM

Prevents the assembling of numeric coprocessor instructions (real or emulated).

%POPLCTL — Ideal, MASM

Resets the listing controls to the way they were when the last **%PUSHLCTL** directive was issued.

PROC — Ideal, MASM

PROC [*language modifier*] [*language*] *name* [*distance*]
    [USES *items*,] [*argument* [,*argument*]...]
    [RETURNS *argument* [,*argument*]...]

name PROC [*language modifier*] [*language*] [*distance*]
    [USES *items*,] [*argument* [,*argument*]...]
    [RETURNS *argument* [,*argument*]...]

Defines the start of procedure *name*. *language modifier* is either **WINDOWS** or **NOWINDOWS**, to specify generation of MSWindows entry/exit code. *language* specifies which language you will be calling from to access this procedure: **C, PASCAL, BASIC, FORTRAN, ASSEMBLER**, or **PROLOG**. This determines symbol naming conventions, the order of any arguments on the stack, and whether the arguments will be left on the stack when the procedure returns. *distance* is **NEAR** or **FAR** and determines the type of

RET instruction that will be assembled at the end of the procedure. *items* is a list of registers and/or single-token data items to be pushed on entry and popped on exit from the procedure. *argument* describes an argument the procedure is called with. Each *argument* has the following syntax:

```
argname[[count1]] [[:distance] [PTR] type] [:count2]
```

argname is the name you'll use to refer to this argument throughout the procedure. *distance* is **NEAR** or **FAR** to indicate that the argument is a pointer of the indicated size. *type* is the data type of the argument and can be **BYTE, WORD, DWORD, FWORD, PWORD, QWORD, TBYTE,** or a structure name. **WORD** is assumed if none is specified. *count1* and *count2* are the number of elements of type. **PTR** tells Turbo Assembler to emit debug information to let Turbo Debugger know that the argument is a pointer to a data item. Using **PTR** without *distance* causes the pointer size to be based on the current memory model and segment address size. **RETURNS** introduces one or more arguments that won't be popped from the stack when the procedure returns.

PUBLIC — Ideal, MASM

PUBLIC [*language*] *symbol* [,[*language*] *symbol*]...

Declares *symbol* to be accessible from other modules. If *language* is specified (**C, PASCAL, BASIC, FORTRAN, ASSEMBLER,** or **PROLOG**), *symbol* is made public after having the naming conventions of the specified language applied to it.

PUBLICDLL — Ideal, MASM

PUBLICDLL [*language*] *symbol* [,[*language*] *symbol*]...

Declares *symbol* to be accessible from other modules as a dynamic link entry point. If *language* is specified (**C, PASCAL, BASIC, FORTRAN, ASSEMBLER,** or **PROLOG**), *symbol* is made public after having the naming conventions of the specified language applied to it.

PURGE — Ideal, MASM

PURGE *macroname* [,*macroname*]...

Removes macro definition *macroname*.

%PUSHLCTL — Ideal, MASM

Saves current listing controls on a 16-level stack.

QUIRKS — Ideal, MASM

Allows you to assemble a source file that makes use of one of the true MASM bugs.

.RADIX — MASM

RADIX — Ideal, MASM

.RADIX *radix*
RADIX *radix*

Sets the default radix for integer constants in expressions to 2, 8, 10, or 16.

RECORD — MASM, Ideal

name RECORD *field* [,*field*]...
RECORD *name field* [,*field*]...

Defines record *name* that contains bit fields. Each *field* describes a group of bits in the record and has the following format:

```
fieldname:width[=expression]
```

fieldname is the name of a field in the record. *width* (1..16) specifies the number of bits in the field. If the total number of bits in all fields is 8 or less, the record will occupy 1 byte; 9..16 bits will occupy 2 bytes; otherwise, it will occupy 4 bytes. *expression* provides a default value for the field.

REPT — Ideal, MASM

REPT *expression*
  statements
ENDM

Repeats a block of statements *expression* times.

.SALL — MASM

Suppresses the listing of all statements in macro expansions.

SEGMENT — MASM, Ideal

SEGMENT *name* [*align*] [*combine*] [*use*] ['*class*']
name SEGMENT [*align*] [*combine*] [*use*] ['*class*']

Defines segment *name* with full attribute control. If you have already defined a segment with the same name, this segment is treated as a continuation of the previous one. *align* specifies the type of memory boundary where the segment must start: **BYTE, WORD, DWORD, PARA** (default), or **PAGE**. *combine* specifies how segments from different modules but with the same name will be combined at link time: **AT** *expression* (locates segment at absolute paragraph address *expression*), **COMMON** (locates this segment and all other segments with the same name at the same address), **MEMORY** (concatenates all segments with the same name to form a single contiguous segment), **PRIVATE** (does not combine this segment with any other segments; this is the default used if none specified), **PUBLIC** (same as **MEMORY** above), or **STACK** (concatenates all segments with the same name to form a single contiguous segment, then initializes SS to the beginning of the segment and SP to the length of the segment. *use* specifies the default word size for the segment if 80386 code generation is enabled, and can be either **USE16** or **USE32**. *class* controls the ordering of segments at link time: segments with the same class name are loaded into memory together, regardless of the order in which they appear in the source file.

.SEQ — MASM

Sets sequential segment-ordering. This is the default ordering mode for Turbo Assembler. **.SEQ** has the same function as the **/s** command-line option.

.SFCOND — MASM

Prevents statements in false conditional blocks from appearing in the listing file.

SIZESTR — Ideal, MASM51

name SIZESTR *string*

Assigns the number of characters in *string* to *name*. A null string has a length of zero.

SMART — MASM

Enables all code optimizations.

.STACK — MASM

STACK — Ideal, MASM

.STACK [*size*]
STACK [*size*]

Defines the start of the stack segment, allocating *size* bytes. 1024 bytes are allocated if *size* is not specified.

STRUC — MASM, Ideal

name STRUC
  fields
[*name*] ENDS

STRUC *name*
  fields
ENDS [*name*]

Defines a structure called *name* containing *fields*. Each field uses the normal data allocation directives (**DB**, **DW**, and so on) to define its size. *fields* may be named or remain nameless. Field names must be unique when using MASM mode but don't need to be when using Ideal mode.

SUBSTR — Ideal, MASM51

name SUBSTR *string*,*position*[,*size*]

Defines a new string *name* consisting of characters from *string* starting at *position*, with a length of *size*. All the remaining characters in *string*, starting from *position*, are assigned to *name* if *size* is not specified.

SUBTTL — MASM

%SUBTTL — Ideal, MASM

SUBTTL *text*
%SUBTTL *"text"*

Sets subtitle in listing file to *text*.

%SYMS — Ideal, MASM

Enables symbol table placement in listing file. This is the default symbol listing mode for Turbo Assembler.

%TABSIZE — Ideal, MASM

%TABSIZE *width*

Sets the number of columns between tabs in the listing file to *width*. The default is 8 columns.

%TEXT — Ideal, MASM

%TEXT *width*
Sets width of source field in listing file to *width* columns.

.TFCOND — MASM

Toggles conditional block-listing mode.

TITLE — MASM

%TITLE — Ideal, MASM

TITLE *text*
%TITLE *"text"*

Sets title in listing file to *text*.

%TRUNC — Ideal, MASM

Truncates listing fields that are too long.

UDATASEG — Ideal, MASM

Defines the start of an uninitialized data segment.

UFARDATA — Ideal, MASM

Defines the start of an uninitialized far data segment.

UNION — Ideal, MASM (disabled by QUIRKS)

UNION *name*
  fields
ENDS [*name*]

name UNION
  fields
[*name*] ENDS

Defines a union called *name*. A union is just like a **STRUC** except that all its members have an offset of zero from the start of the union. This results in a set of fields that are overlayed, allowing you to refer to the memory area defined by the union with different names and different data sizes. The length of a union is the length of its largest member, not the sum of the lengths of its members as in a **STRUC**. *fields* define the fields that comprise the union. Each field uses the normal data allocation directives (**DB**, **DW**, etc.) to define its size.

USES — Ideal, MASM

USES *item* [,*item*]...

Indicates which registers or single-token data items you want to have pushed at the beginning of the enclosing procedure and which ones you want popped just before the procedure returns. You must use this directive before the first instruction that actually generates code in your procedure.

WARN — Ideal, MASM

WARN [*warnclass*]

Enables the type of warning message specified with *warnclass*, or all warnings if *warnclass* is not specified. *warnclass* may be one of: **ALN, ASS, BRK, ICG, LCO, OPI, OPP, OPS, OVF, PDC, PRO, PQK, RES,** or **TPI.**

.XALL — MASM

Causes only macro expansions that generate code or data to be listed.

.XCREF — MASM

Disables cross-reference listing (CREF) information accumulation.

.XLIST — MASM

Disables subsequent output to listing file.

P A R T 4

Processor Instructions

This chapter presents instructions for the x86 in alphabetical order. For each instruction, the forms are given for each operand combination, including object code produced, operands required, execution time, and a description. For each instruction, there is an operational description and a summary of exceptions generated.

Operand-size and address-size attributes

When executing an instruction, the x86 can address memory using either 16- or 32-bit addresses. Consequently, each instruction that uses memory addresses has associated with it an address-size attribute of either 16 or 32 bits. Sixteen-bit addresses imply both the use of a 16-bit displacement in the instruction and the generation of a 16-bit address offset (segment relative address) as the result of the effective address calculation. Thirty-two-bit addresses imply the use of a 32-bit displacement and the generation of a 32-bit address offset. Similarly, an instruction that accesses words (16 bits) or doublewords (32 bits) has an operand-size attribute of either 16 or 32 bits.

The attributes are determined by a combination of defaults, instruction prefixes, and (for programs executing in protected mode) size-specification bits in segment descriptors.

Default segment attribute

For programs executed in protected mode, the D-bit in executable-segment descriptors determines the default attribute for both address size and operand size. These default attributes apply to the execution of all instructions in the segment. A value of zero in the D-bit sets the default address size and operand size to 16 bits; a value of one, to 32 bits.

Programs that execute in real mode or virtual-8086 mode have 16-bit addresses and operands by default.

Operand-size and address-size instruction prefixes

The internal encoding of an instruction can include two byte-long prefixes: the addres-size prefix, 67H, and the operand-size prefix, 66H. (A later section, "Instruction format," shows the position of the prefixes in an instruction's encoding.) These prefixes *override* the default segment attributes for the instruction that follows. Table 4.1 shows the effect of each possible combination of defaults and overrides.

Table 4.1 Effective size attributes

Segment default D= ...	0	0	0	0	1	1	1	1
Operand-size prefix 66h	N	N	Y	Y	N	N	Y	Y
Address-size prefix 67h	N	Y	N	Y	N	Y	N	Y
Effective operand size	16	16	32	32	32	32	16	16
Effective address size	16	32	16	32	32	16	32	16

Y = Yes, this instruction prefix is present.
N = No, this instruction prefix is *not* present.

Address-size attribute for stack

Instructions that use the stack implicitly (for example, POP EAX) also have a stack address-size attribute of either 16 or 32 bits. Instructions with a stack address-size attribute of 16 use the 16-bit SP stack pointer register; instructions with a stack address-size attribute of 32 bits use the 32-bit ESP register to form the address of the top of the stack.

The stack address-size attribute is controlled by the B-bit of the data-segment descriptor in the SS register. A value of zero in the B-bit selects a stack address-size attribute of 16; a value of one selects a stack address-size attribute of 32.

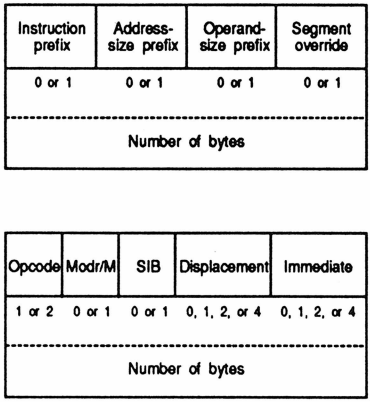
Instruction format

All instruction encodings are subsets of the general instruction format shown in Figure 4.1. Instructions consist of optional instruction prefixes, one or two primary opcode bytes, possibly an address specifier consisting of the ModR/M byte and the SIB (Scale Index Base) byte, a displacement, if required, and an immediate data field, if required.

Smaller encoding fields can be defined within the primary opcode or opcodes. These fields define the direction of the operation, the size of the displacements, the register encoding, or sign extension; encoding fields vary depending on the class of operation.

Most instructions that can refer to a operand in memory have an addressing form byte following the primary opcode byte(s). This byte, called the ModR/M byte, specifies the address form to be used. Certain encodings of the ModR/M byte indicate a second addressing byte, the SIB byte, which follows the ModR/M byte and is required to fully specify the addressing form.

Figure 4.1
80386 instruction format



Addressing forms can include a displacement immediately following either the ModR/M or SIB byte. If a displacement is present, it can be 8, 16, or 32 bits.

If the instruction specifies an immediate operand, the immediate operand always follows any displacement bytes. The immediate operand, if specified, is always the last field of the instruction.

The following are the allowable instruction prefix codes:

- F3h: REP prefix (used only with string instructions)
- F3h: REPE/REPZ prefix (used only with string instructions)
- F2h: REPNE/REPNZ prefix (used only with string instructions)
- F0h: LOCK prefix

The following are the segment override prefixes:

- 2Eh: CS segment override prefix
- 36h: SS segment override prefix
- 3Eh: DS segment override prefix
- 26h: ES segment override prefix
- 64h: FS segment override prefix (386 only)
- 65h: GS segment override prefix (386 only)
- 66h: Operand-size override
- 67h: Address-size operand

ModR/M and SIB bytes

The ModR/M and SIB bytes follow the opcode byte(s) in many of the x86 instructions. They contain the following information: the indexing type or register number to be used in the instruction; the register to be used, or more information to select the instruction; and the base, index, and scale information.

The ModR/M byte contains three fields of information:

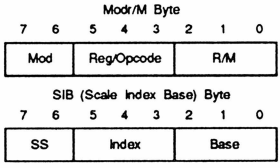
- The **mod** field, which occupies the two most significant bits of the byte, combines with the r/m field to form 32 possible values: 8 registers and 24 indexing modes.
- The **reg** field, which occupies the next three bits following the mod field, specifies either a register number or three more bits of opcode information. The meaning of the reg field is determined by the first (opcode) byte of the instruction.
- The **r/m** field, which occupies the three least-significant bits of the byte, can specify a register as the location of an operand, or can form part of the addressing-mode encoding in combination with the **mod** field as described earlier.

The based indexed and scaled indexed forms of 32-bit addressing require the SIB byte. The presence of the SIB byte is indicated by certain encodings of the ModR/M byte. The SIB byte then includes the following fields:

- The **ss** field, which occupies the 2 most-significant bits of the byte, specifies the scale factor.
- The **index** field, which occupies the next 3 bits following the **ss** field specifies the register number of the index register.
- The **base** field, which occupies the 3 least-significant bits of the byte, specifies the register number of the base register.

Figure 4.2 shows the format of the ModR/M and SIB bytes.

Figure 4.2
ModR/M and SIB byte formats



The values and corresponding addressing forms of the ModR/M and SIB bytes are shown in Tables 4.2, 4.3, and 4.4.

Table 4.2 16-bit addressing forms with ModR/M byte

r8(/r)		AL	CL	DL	BL	AH	CH	DH	BH
r16(/r)		AX	CX	DX	BX	SP	BP	SI	DI
r32(/r)		EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
/digit (opcode)		0	1	2	3	4	5	6	7
REG =		000	001	010	011	100	101	110	111
Effective Address	**ModR/M**	**ModR/M values in hexadecimal**							
[BX + SI]		000	00	08	10	18	20	28	30	38
[BX + DI]		001	01	09	11	19	21	29	31	39
[BP + SI]		010	02	0A	12	1A	22	2A	32	3A
[BP + DI]	00	011	03	0B	13	1B	23	2B	33	3B
[SI]		100	04	0C	14	1C	24	2C	34	3C
[DI]		101	05	0D	15	1D	25	2D	35	3D
disp16		110	06	0E	16	1E	26	2E	36	3E
[BX]		111	07	0F	17	1F	27	2F	37	3F
[BX + SI] + disp8		000	40	48	50	58	60	68	70	78
[BX + DI] + disp8		001	41	49	51	59	61	69	71	79
[BP + SI] + disp8		010	42	4A	52	5A	62	6A	72	7A
[BP + DI] + disp8	01	011	43	4B	53	5B	63	6B	73	7B
[SI] + disp8		100	44	4C	54	5C	64	6C	74	7C
[DI] + disp8		101	45	4D	55	5D	65	6D	75	7D
[BP] + disp8		110	46	4E	56	5E	66	6E	76	7E
[BX] + disp8		111	47	4F	57	5F	67	6F	77	7F
[BX + SI] + disp16		000	80	88	90	98	A0	A8	B0	B8
[BX + DI] + disp16		001	81	89	91	99	A1	A9	B1	B9
[BP + SI] + disp16		010	82	8A	92	9A	A2	AA	B2	BA
[BP + DI] + disp16	10	011	83	8B	93	9B	A3	AB	B3	BB
[SI] + disp16		100	84	8C	94	9C	A4	AC	B4	BC
[DI] + disp16		101	85	8D	95	9D	A5	AD	B5	BD
[BP] + disp16		110	86	8E	96	9E	A6	AE	B6	BE
[BX] + disp16		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL (386)		000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL (386)		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL (386)		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL (386)	11	011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH (386)		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH (386)		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH (386)		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH (386)		111	C7	CF	D7	DF	E7	EF	F7	FF

disp8 denotes an 8-bit displacement following the ModR/M byte, to be sign-extended and added to the index. *disp16* denotes a 16-bit displacement following the ModR/M byte, to be added to the index. Default segment register is SS for the effective addresses containing a BP index, DS for other effective addresses.

Table 4.3 32-bit addressing forms with ModR/M byte (386 only)

r8(/r)		AL	CL	DL	BL	AH	CH	DH	BH
r16(/r)		AX	CX	DX	BX	SP	BP	SI	DI
r32(/r)		EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
/digit(opcode)		0	1	2	3	4	5	6	7
REG =		000	001	010	011	100	101	110	111
Effective Address	**ModR/M**	**ModR/M values in hexadecimal**							
[EAX]	000	00	08	10	18	20	28	30	38
[ECX]	001	01	09	11	19	21	29	31	39
[EDX]	010	02	0A	12	1A	22	2A	32	3A
[EBX]	00 011	03	0B	13	1B	23	2B	33	3B
[- -] [- -]	100	04	0C	14	1C	24	2C	34	3C
disp32	101	05	0D	15	1D	25	2D	35	3D
[ESI]	110	06	0E	16	1E	26	2E	36	3E
[EDI]	111	07	0F	17	1F	27	2F	37	3F
disp8[EAX]	000	40	48	50	58	60	68	70	78
disp8[ECX]	001	41	49	51	59	61	69	71	79
disp8[EDX]	010	42	4A	52	5A	62	6A	72	7A
disp8[EPX];	01 011	43	4B	53	5B	63	6B	73	7B
disp8[- -] [- -]	100	44	4C	54	5C	64	6C	74	7C
disp8[EBP]	101	45	4D	55	5D	65	6D	75	7D
disp8[ESI]	110	46	4E	56	5E	66	6E	76	7E
disp8[EDI]	111	47	4F	57	5F	67	6F	77	7F
disp32[EAX]	000	80	88	90	98	A0	A8	B0	B8
disp32[ECX]	001	81	89	91	99	A1	A9	B1	B9
disp32[EDX]	010	82	8A	92	9A	A2	AA	B2	BA
disp32[EBX]	10 011	83	8B	93	9B	A3	AB	B3	BB
disp32[- -] [- -]	100	84	8C	94	9C	A4	AC	B4	BC
disp32[EBP]	101	85	8D	95	9D	A5	AD	B5	BD
disp32[ESI]	110	86	8E	96	9E	A6	AE	B6	BE
disp32[EDI]	111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL	001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL	010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL	11 011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH	100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH	101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH	110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH	111	C7	CF	D7	DF	E7	EF	F7	FF

[- -] [- -] means a SIB follows the ModR/M byte. *disp8* denotes an 8-bit displacement following the SIB byte, to be sign-extended and added to the index. *disp32* denotes a 32-bit displacement following the ModR/M byte, to be added to the index.

Table 4.4 32-bit addressing forms with SIB byte (386 only)

r32		EAX	ECX	EDX	EBX	ESP	[*]	ESI	EDI
Base =		0	1	2	3	4	5	6	7
Base =		000	001	010	011	100	101	110	111
Scaled index	**SS index**	**ModR/M values in hexadecimal**							
[EAX]		000	00	01	02	03	04	05	06	07
[ECX]		001	08	09	0A	0B	0C	0D	0E	0F
[EDX]		010	10	11	12	13	14	15	16	17
[EBX]	00	011	18	19	1A	1B	1C	1D	1E	1F
none		100	20	21	22	23	24	25	26	27
[EBP]		101	28	29	2A	2B	2C	2D	2E	2F
[ESI]		110	30	31	32	33	34	35	36	37
[EDI]		111	38	39	3A	3B	3C	3D	3E	3F
[EAX*2]		000	40	41	42	44	44	45	46	47
[ECX*2]		001	48	49	4A	4B	4C	4D	4E	4F
[ECX*2]		010	50	51	52	55	54	55	56	57
[EBX*2]	01	011	58	59	5A	5B	5C	5D	5E	5F
none		100	60	61	62	63	64	65	66	67
[EBP*2]		101	68	69	6A	6B	6C	6D	6E	6F
[ESI*2]		110	70	71	72	73	74	75	76	77
[EDI*2]		111	78	79	7A	7B	7C	7D	7E	7F
[EAX*4]		000	80	81	82	83	84	85	86	87
[ECX*4]		001	88	89	8A	8B	8C	8D	8E	8F
[EDX*4]		010	90	91	92	93	94	95	96	97
[EBX*4]	10	011	98	89	9A	9B	9C	9D	9E	9F
none		100	A0	A1	A2	A3	A4	A5	A6	A7
[EBP*4]		101	A8	A9	AA	AB	AC	AD	AE	AF
[ESI*4]		110	B0	B1	B2	B3	B4	B5	B6	B7
[EDI*4]		111	B8	B9	BA	BB	BC	BD	BE	BF
[EAX*8]		000	C0	C1	C2	C3	C4	C5	C6	C7
[ECX*8]		001	C8	C9	CA	CB	CC	CD	CE	CF
[EDX*8]		010	D0	D1	D2	D3	D4	D5	D6	D7
[EBX*8]	11	011	D8	D9	DA	DB	DC	DD	DE	DF
none		100	E0	E1	E2	E3	E4	E5	E6	E7
[EBP*8]		101	E8	E9	EA	EB	EC	ED	EE	EF
[ESI*8]		110	F0	F1	F2	F3	F4	F5	F6	F7
[EDI*8]		111	F8	F9	FA	FB	FC	FD	FE	FF

[*] means a disp32 with no base if MOD is 00; otherwise, [ESP]. This provides the following addressing modes:

disp32[index]	(MOD=00)
disp8[EBP][index]	(MOD=01)
disp32[EBP][index]	(MOD=10)

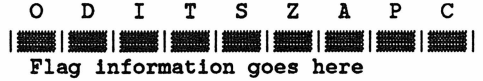
How to read the instruction set pages

Here's a sample of the format of this chapter:

Instruction name

What the instruction name means

What processor the instruction works on



Opcode	Instruction	Clocks		
		386	286	86

This table contains clock information.

Flags

Each entry in this section includes information on which flags in the x86's flag register are changed and how. Each flag has a one-letter tag for its name.

- **O** = Overflow flag
- **D** = Direction flag
- **I** = Interrupt flag
- **T** = Trap flag
- **S** = Sign flag
- **Z** = Zero flag
- **A** = Auxiliary flag
- **P** = Parity flag
- **C** = Carry flag

The following symbols indicate how the flag register has changed:

- **?** = Undefined after the operation
- ***** = Changed to reflect the results of the instruction.
- **0** = Always cleared
- **1** = Always set

Opcode

The "Opcode" column gives the complete object code produced for each form of the instruction. When possible, the codes are given as hexadecimal bytes, in the same order in which they appear in memory. Definitions of entries other than hexadecimal bytes are as follows:

/digit

(digit is between 0 and 7.) Indicates that the ModR/M byte of the instruction uses only the r/m (register or memory) operand. The reg field contains the digit that provides an extension to the instruction's opcode.

/r

Indicates that the ModR/M byte of the instruction contains both a register operand and an r/m operand.

cb, cw, cd, cp

A 1-byte (cb), 2-byte (cw), 4-byte (cd), or 6-byte (cp) value following the opcode that is used to specify a code offset and possibly a new value for the code segment register.

ib, iw, id

A 1-byte (ib), 2-byte (iw), or 4-byte (id) immediate operand to the instruction that follows the opcode, ModR/M bytes, or scale-indexing bytes. The opcode determines if the operand is a signed value. All words and doublewords are given with the low-order byte first.

+rb, +rw, +rd

A register code, from 0 through 7, added to the hexadecimal byte given at the left of the plus sign to form a single opcode byte. The codes are

rb	rw	rd (386)
AL = 0	AX = 0	EAX = 0
CL = 1	CX = 1	ECX = 1
DL = 2	DX = 2	EDX = 2
BL = 3	BX = 3	EBX = 3
AH = 4	SP = 4	ESP = 4
AH = 4	SP = 4	ESP = 4
CH = 5	BP = 5	EBP = 5

DH = 6 SI = 6 ESI = 6
BH = 7 DI = 7 EDI = 7

Instruction

The "Instruction" column gives the syntax of the instruction statement as it would appear in a TASM 386 program. The following is a list of the symbols used to represent operands in the instruction statements:

rel8

A relative address in the range from 128 bytes before the end of the instruction to 127 bytes after the end of the instruction.

rel16, rel32

A relative address within the same code segment as the instruction assembled. **rel16** applies to instructions with an operand-size attribute of 16 bits; **rel32** applies to instructions with an operand-size attribute of 32 bits (80386 only).

ptr16:16, ptr16:32

A far pointer, typically in a code segment different from that of the instruction. The notation **16:16** indicates that the value of the pointer has two parts. The value to the right of the colon is a 16-bit selector or value destined for the code segment register. The value to the left corresponds to the offset within the destination segment. **ptr16:16** is used when the instruction's operand-size attribute is 16 bits; **ptr16:32** is used with the 32-bit attribute (80386 only).

r8

One of the byte registers AL, CL, DL, BL, AH, CH, DH, or BH.

r16

One of the word registers AX, CX, DX, BX, SP, BP, SI, or DI.

r32 (80386)

One of the doubleword registers EAX, ECX, EDX, EBX, ESP, EBP, ESI, or EDI.

imm8

An immediate byte value. **imm8** is a signed number between –128 and +127 inclusive. For instructions in which **imm8** is combined with a word or doubleword operand, the immediate value is sign-extended to form a word or doubleword. The upper byte of the word is filled with the topmost bit of the immediate value.

imm16

An immediate word value used for instructions whose operand-size attribute is 16 bits. This is a number between –32,768 and +32,767 inclusive.

imm32 (80386)

An immediate doubleword value used for instructions whose operand-size attribute is 32 bits. It allows the use of a number between +2,147,483,647 and –2,147,483,648.

r/m8

A 1-byte operand that is either the contents of a byte register (AL, BL, CL, DL, AH, BH, CH, DH), or a byte from memory.

r/m16

A word register or memory operand used for instructions whose operand-size attribute is 16 bits. The word registers are AX, BX, CX, DX, SP, BP, SI, DI. The contents of memory are found at the address provided by the effective address computation.

r/m32

A doubleword register or memory operand used for instructions whose operand-size attribute is 32 bits. The doubleword registers are EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI. The contents of memory are found at the address provided by the effective address computation.

m8

A memory byte addressed by DS:SI or ES:DI (used only by string instructions on the 80386).

m16

A memory word addressed by DS:SI or ES:DI (used only by string instructions).

m32

A memory doubleword addressed by DS:SI or ES:DI (used only by string instructions).

m16:16, m16:32 (80386)

A memory operand containing a far pointer composed of two numbers. The number to the left of the colon corresponds to the pointer's segment selector. The number to the right corresponds to its offset.

m16 & 32, m16 & 16 (186/286/386), m32 & 32 (80386)

A memory operand consisting of data item pairs whose sizes are indicated on the left and the right side of the ampersand. All memory addressing modes are allowed. **m16 & 16** and **m32 & 32** operands are used by the BOUND instruction to provide an operand containing an upper and lower bounds for array indices. **m16 & 32** is used by LIDT and LGDT to provide a word with which to load the limit field, and a doubleword with which to load the base field of the corresponding Global and Interrupt Descriptor Table Registers.

moffs8, moffs16, moffs32 (memory offset; 386 only)

A simple memory variable of type **BYTE**, **WORD**, or **DWORD** (386) used by some variants of the MOV instruction. The actual address is given by a simple offset relative to the segment base. No ModR/M byte is used in the instruction. The number shown with **moffs** indicates its size, which is determined by the address-size attribute of the instruction.

Sreg

A segment register. The segment register bit assignments are ES = 0, CS = 1, SS = 2, DS = 3, FS = 4 (386), and GS = 5 (386).

Clocks

The "Clocks" column gives the number of clock cycles the instruction takes to execute. The clock count calculations make the following assumptions:

- The instruction has been prefetched and decoded and is ready for execution.
- Bus cycles do not require wait states.
- There are no local bus HOLD requests delaying processor access to the bus.
- No exceptions are detected during instruction execution.
- Memory operands are aligned.

Clock counts for instructions that have an r/m (register or memory) operand are separated by a slash. The count to the left is used for a register operand; the count to the right is used for a memory operand.

The following symbols are used in the clock count specifications:

- **n**, which represents a number of repetitions.
- **m**, which represents the number of components in the next instruction executed, where the entire displacement (if any) counts as one component, the entire immediate data (if any) counts as one component, and every other byte of the instruction and prefix(es) each counts as one component.
- **pm=**, a clock count that applies when the instruction executes in protected mode. **pm=** is not given when the clock counts are the same for protected and real address modes.

When an exception occurs during the execution of an instruction and the exception handler is in another task, the instruction exception time is increased by the number of clocks to effect a task switch. This parameter depends on several factors:

- The type of TSS used to represent the current task (386 TSS or 286 TSS).
- The type of TSS used to represent the new task.
- Whether the current task is in V86 mode.
- Whether the new task is in V86 mode.

Note: Users should read Intel's documentation for more information about protected mode and task switching.

AAA

ASCII Adjust after Addition

O	D	I	T	S	Z	A	P	C

		?			?	?	*	?	*

Opcode	Instruction	Clocks		
		386	286	86
37	AAA	4	3	8

Execute AAA only following an ADD instruction that leaves a byte result in the AL register. The lower nibbles of the operands of the ADD instruction should be in the range 0 through 9 (BCD digits). In this case, AAA adjusts AL to contain the correct decimal digit result. If the addition produced a decimal carry, the AH register is incremented, and the carry and auxiliary carry flags are set to 1. If there was no decimal carry, the carry and auxiliary flags are set to 0 and AH is unchanged. In either case, AL is left with its top nibble set to 0. To convert AL to an ASCII result, follow the AAA instruction with OR AL, 30H.

AAD

ASCII Adjust AX before Division

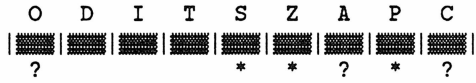
O	D	I	T	S	Z	A	P	C
?				*	*	?	*	?

Opcode	Instruction	Clocks		
		386	286	86
D5 0A	AAD	19	14	60

AAD is used to prepare two unpacked BCD digits (the least-significant digit in AL, the most-significant digit in AH) for a division operation that will yield an unpacked result. This is accomplished by setting AL to AL + (10 * AH), and then setting AH to 0. AX is then equal to the binary equivalent of the original unpacked two-digit number.

AAM

ASCII Adjust AX after Multiply

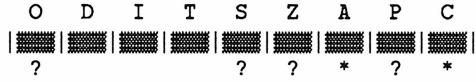


Opcode	Instruction	Clocks		
		386	286	86
D4 0A	AAM	17	16	83

Execute AAM only after executing a MUL instruction between two unpacked BCD digits that leaves the result in the AX register. Because the result is less than 100, it is contained entirely in the AL register. AAM unpacks the AL result by dividing AL by 10, leaving the quotient (most-significant digit) in AH and the remainder (least-significant digit) in AL.

AAS

ASCII Adjust AL After Subtraction

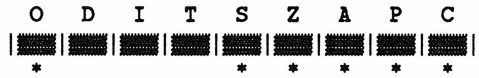


Opcode	Instruction	Clocks		
		386	286	86
3F	AAS	4	3	8

Execute AAS only after a SUB instruction that leaves the byte result in the AL register. The lower nibbles of the operands of the SUB instruction must have been in the range 0 through 9 (BCD digits). In this case, AAS adjusts AL so it contains the correct decimal digit result. If the subtraction produced a decimal carry, the AH register is decremented, and the carry and auxiliary carry flags are set to 1. If no decimal carry occurred, the carry and auxiliary carry flags are set to 0, and AH is unchanged. In either case, AL is left with its top nibble set to 0. To convert AL to an ASCII result, follow the AAS with OR AL, 30H.

ADC

Add with Carry



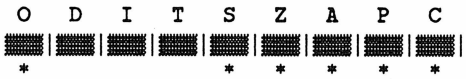
Opcode	Instruction		Clocks		Description
		386	286	86	
10 */r*	ADC *r/m8,r8*	2/7	2/7	3/16+EA	Add with carry byte register to *r/m* byte
11 */r*	ADC *r/m16,r16*	2/7	2/7	3/16+EA	Add with carry word register to *r/m* word
11 */r*	ADC *r/m32,r32*	2/7			Add with CF dword register to *r/m* word
12 */r*	ADC *r8,r/m8*	2/6	2/7	3/9+EA	Add with carry *r/m* byte to byte register
13 */r*	ADC *r16,r/m16*	2/6	2/7	3/9+EA	Add with carry *r/m* word to word register
13 */r*	ADC *r32,r/m32*	2/6			Add with CF *r/m* dword to dword register
14 *ib*	ADC AL,*imm8*	2	3	4	Add with carry immediate byte to AL
15 *iw*	ADC AX,*imm16*	2	3	4	Add with carry immediate word to AX
15 *id*	ADC EAX,*imm32*	2			Add with carry immediate dword to EAX
80 */2 ib*	ADC *r/m8,imm8*	2/7	3/7	4/17+EA	Add with carry immediate byte to *r/m* byte
81 */2 iw*	ADC *r/m16,imm16*	2/7	3/7	4/17+EA	Add with carry immediate word to *r/m* word
81 */2 id*	ADC *r/m32,imm32*	2/7			Add with CF immediate dword to *r/m* dword
83 */2 ib*	ADC *r/m16,imm8*	2/7	3/7	4/17+EA	Add with CF sign-extended immediate byte to *r/m* word
83 */2 ib*	ADC *r/m32,imm8*	2/7			Add with CF sign-extended immediate byte into *r/m* dword

ADC performs an integer addition of the two operands DEST and SRC and the carry flag, CF. The result of the addition is assigned to the first operand (DEST), and the flags are set accordingly. ADC is usually executed as part of a multi-byte or multi-word addition operation. When an immediate byte value is added to a word or doubleword operand, the

immediate value is first sign-extended to the size of the word or doubleword operand.

ADD

Add



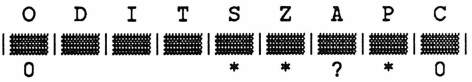
Opcode	Instruction	Clocks			Description
		386	286	86	
04 *ib*	ADD AL,*imm8*	2	3	4	Add immediate byte to AL
05 *iw*	ADD AX,*imm16*	2	3	4	Add immediate word to AX
05 *id*	ADD EAX,*imm32*	2			Add immediated dword to EAX
80 /0 *ib*	ADD *r/m8,imm8*	2/7	3/7	4/17+EA	Add immediate byte to *r/m* byte
81 /0 *iw*	ADD *r/m16,imm16*	2/7	3/7	4/17+EA	Add immediate word to *r/m* word
81 /0 *id*	ADD *r/m32,imm32*	2/7			Add immediate dword to *r/m* dword
83 /0 *ib*	ADD *r/m16,imm8*	2/7	3/7	4/17+EA	Add sign-extended immediate byte to *r/m* word
83 /0 *ib*	ADD *r/m32,imm8*	2/7			Add sign-extended immediate byte to *r/m* dword
00 /*r*	ADD *r/m8,r8*	2/7	2/7	3/16+EA	Add byte register to *r/m* byte
01 /*r*	ADD *r/m16,r16*	2/7	2/7	3/16+EA	Add word register to *r/m* word
01 /*r*	ADD *r/m32,r32*	2/7			Add dword register to *r/m* dword
02 /*r*	ADD *r8,r/m8*	2/6	2/7	3/9+EA	Add *r/m* byte to byte register
03 /*r*	ADD *r16,r/m16*	2/6	2/7	3/9+EA	Add *r/m* word to word register
03 /*r*	ADD *r32,r/m32*	2/6			Add *r/m* dword to dword register

ADD performs an integer addition of the two operands (DEST and SRC). The result of the addition is assigned to the first operand (DEST), and the flags are set accordingly.

When an immediate byte is added to a word or doubleword operand, the immediate value is sign-extended to the size of the word or doubleword operand.

AND

Logical AND



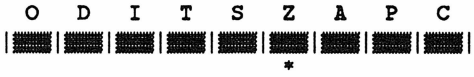
Opcode	Instruction	Clocks			Description
		386	286	86	
20 /r	AND *r/m8,r8*	2/7	2/7	3/16+EA	AND byte register into *r/m* byte
21 /r	AND *r/m16,r16*	2/7	2/7	3/16+EA	AND word register into *r/m* word
21 /r	AND *r/m32,r32*	2/7			AND dword register to *r/m* dword
22 /r	AND *r8,r/m8*	2/6	2/7	3/9+EA	AND *r/m* byte to byte register
23 /r	AND *r16,r/m16*	2/6	2/7	3/9+EA	AND *r/m* word to word register
23 /r	AND *r32,r/m32*	2/6			AND *r/m* dword to dword register
24 *ib*	AND AL,*imm8*	2	3	4	AND immediate byte to AL
25 *iw*	AND AX,*imm16*	2	3	4	AND immediate word to AX
25 *id*	AND EAX,*imm32*	2			AND immediate dword to EAX
80 /4 *ib*	AND *r/m8,imm8*	2/7	3/7	4/17+EA	AND immediate byte to *r/m* byte
81 /4 *iw*	AND *r/m16,imm16*	2/7	3/7	4/17+EA	AND immediate word to *r/m* word
81 /4 *id*	AND *r/m32,imm32*	2/7			AND immediate dword to *r/m* word
83 /4 *ib*	AND *r/m16,imm8*	2/7			AND sign-extended immediate byte with *r/m* word
83 /4 *ib*	AND *r/m32,imm8*	2/7			AND sign-extended immediate byte with *r/m* dword

Each bit of the result of the AND instruction is a 1 if both corresponding bits of the operands are 1; otherwise, it becomes a 0.

ARPL

Adjust RPL Field of Selector

80286/80386 protected mode only



Opcode	Instruction	Clocks			Description
		386	286	86	
63 /r	ARPL *r/m16,r16*	*pm*=20/21	*pm*=10/11		Adjust RPL of *r/m*16 to not less than RPL of r16

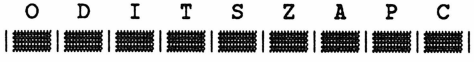
The ARPL instruction has two operands. The first operand is a 16-bit memory variable or word register that contains the value of a selector. The second operand is a word register. If the RPL field ("requested privilege level" —bottom two bits) of the first operand is less than the RPL field of the second operand, the zero flag is set to 1 and the RPL field of the first operand is increased to match the second operand. Otherwise, the zero flag is set to 0 and no change is made to the first operand.

ARPL appears in operating system software, not in application programs. It is used to guarantee that a selector parameter to a subroutine does not request more privilege than the caller is allowed. The second operand of ARPL is normally a register that contains the CS selector value of the caller.

BOUND

Check Array Index Against Bounds

80186/286/386 only



Opcode	Instruction	Clocks			Description
		386	286	86	

62 /r	BOUND *r16, m16&16*	10	13	Check if r16 is within bounds(passes test)
62 /r	BOUND *r32, m32&32*	10		Check if r32 is within bounds(passes test)

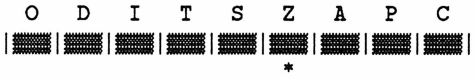
BOUND ensures that a signed array index is within the limits specified by a block of memory consisting of an upper and a lower bound. Each bound uses one word for an operand-size attribute of 16 bits and a doubleword for an operand-size attribute of 32 bits. The first operand (a register) must be greater than or equal to the first bound in memory (lower bound), and less than or equal to the second bound in memory (upper bound). If the register is not within bounds, an Interrupt 5 occurs; the return EIP points to the the BOUND instruction.

The bounds limit data structure is usually placed just before the array itself, making the limits addressable via a constant offset from the beginning of the array.

BSF

Bit Scan Forward

80386 only



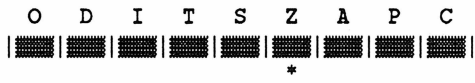
Opcode	Instruction	Clocks			Description
		386	286	86	
0F BC	BSF *r16,r/m16*	10+3*n*			Bit scan forward on *r/m* word
0F BC	BSF *r32,r/m32*	10+3*n*			Bit scan forward on *r/m* dword

BSF scans the bits in the second word or doubleword operand starting with bit 0. The ZF flag is cleared if the bits are all 0; otherwise, the ZF flag is set and the destination register is loaded with the bit index of the first set bit.

BSR

Bit Scan Reverse

80386 only



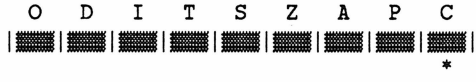
Opcode	Instruction	Clocks			Description
		386	286	86	
0F BD	BSR *r16,r/m16*	10+3*n*			Bit scan reverse on *r/m* word
0F BD	BSR *r32,r/m32*	10+3*n*			Bit scan reverse on *r/m* dword

BSR scans the bits in the second word or doubleword operand from the most significant bit to the least significant bit. The ZF flag is cleared if the bits are all 0; otherwise, ZF is set and the destination register is loaded with the bit index of the first set bit found when scanning in the reverse direction.

BT

Bit Test

80386 only



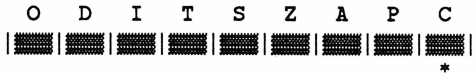
Opcode	Instruction	Clocks			Description
		386	286	86	
0F A3	BT *r/m16,r16*	3/12			Save bit in carry flag
0F A3	BT *r/m32,r32*	3/12			Save bit in carry flag
0F BA / 4 *ib*	BT *r/m16,imm8*	3/6			Save bit in carry flag
0F BA / 4 *ib*	BT *r/m32,imm8*	3/6			Save bit in carry flag

BT saves the value of the bit indicated by the base (first operand) and the bit offset (second operand) into the carry flag.

BTC

Bit Test and Complement

80386 only



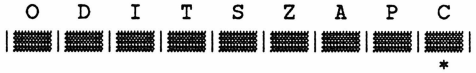
Opcode	Instruction	Clocks			Description
		386	286	86	
0F BB	BTC *r/m16,r16*	6/13			Save bit in carry flag and complement
0F BB	BTC *r/m32,r32*	6/13			Save bit in carry flag and complement
0F BA / *7 ib*	BTC *r/m16,imm8*	6/8			Save bit in carry flag and complement
0F BA / *7 ib*	BTC *r/m32,imm8*	6/8			Save bit in carry flag and complement

BTC saves the value of the bit indicated by the base (first operand) and the bit offset (second operand) into the carry flag and then complements the bit.

BTR

Bit Test and Reset

80386 only



Opcode	Instruction	Clocks			Description
		386	286	86	
0F B3	BTR *r/m16,r16*	6/13			Save bit in carry flag and reset
0F B3	BTR *r/m32,r32*	6/13			Save bit in carry flag and reset
0F BA / *6 ib*	BTR *r/m16,imm8*	6/8			Save bit in carry flag and reset

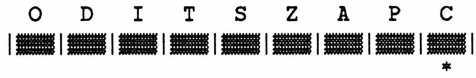
0F BA / *6 ib*	BTR *r/m32,imm8*	6/8	Save bit in carry flag and reset

BTR saves the value of the bit indicated by the base (first operand) and the bit offset (second operand) into the carry flag and then stores 0 in the bit.

BTS

Bit Test and Set

80386 only

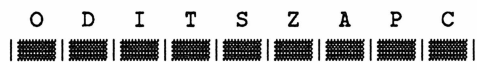


Opcode	Instruction	386	286	86	Description
0F AB	BTS *r/m16,r16*	6/13			Save bit in carry flag and set
0F AB	BTS *r/m32,r32*	6/13			Save bit in carry flag and set
0F BA / *5 ib*	BTS *r/m16,imm8*	6/8			Save bit in carry flag and set
0F BA / *5 ib*	BTS *r/m32,imm8*	6/8			Save bit in carry flag and set

BTS saves the value of the bit indicated by the base (first operand) and the bit offset (second operand) into the carry flag and then stores 1 in the bit.

CALL

Call Procedure



All flags are affected if a task switch occurs; no flags are affected if a task switch does not occur.

Opcode	Instruction	386	286*	86	Description

E8 *cw*	CALL *rel16*	7+*m*	7	19	Call near,displacement relative to next instruction
FF /2	CALL *r/m16*	7+*m*/10+*m*	7/11	16/21+EA	Call near, register indirect/ memory indirect
9A *cd*	CALL *ptr16:16*	17+*m*,*pm*=34 =*m*	13,*pm* =26	28	Call intersegment, to full pointer given
9A *cd*	CALL *ptr16:16*	*pm*=52+*m*	41		Call gate, same privilege
9A *cd*	CALL *ptr16:16*	*pm*=86+*m*	82		Call gate, more privilege, no parameters
9A *cd*	CALL *ptr16:16*	*pm*=94+4*x* +*m*	86+4*x*		Call gate, more privilege, X parameters
9A *cd*	CALL *ptr16:16*	ts	177/182		Call to task (via task state segment/task gate for 286)
FF /3	CALL *m16:16*	22+*m*,*pm*38 +*m*	16/29	37+EA	Call intersegment, address at *r/m* dword
FF /3	CALL *m16:16*	*pm*=56+*m*	44		Call gate, same privilege
FF /3	CALL *m16:16*	*pm*=90+*m*	83		Call gate, more privilege, no parameters
FF /3	CALL *m16:16*	*pm*=98+4*x* +*m*	90+4*x*		Call gate, more privilege, X parameters
FF /3	CALL *m16:16*	5 + ts	180/185		Call to task (Via task state segment/task gate for 286)
E8 *cd*	CALL *rel32*	7+*m*			Call near, displacement relative to next instruction
FF /2	CALL *r/m32*	7+*m*/10+*m*			Call near,indirect
9A *cp*	CALL *ptr16:32*	17+*m*,*pm*= 34+*m*			Call intersegment, to full pointer given
9A *cp*	CALL *ptr16:32*	*pm*=52+*m*			Call gate, same privilege
9A *cp*	CALL *ptr16:32*	*pm*=86+*m*			Call gate, more privilege, no parameters
9A *cp*	CALL *ptr32:32*	*pm*=94+4*x*+*m*			Call gate, more privilege,x parameters
9A *cp*	CALL *ptr16:32*	ts			Call to task
FF /3	CALL *m16:32*	22+*m*,*pm*= 38+*m*			Call intersegment, address at *r/m* dword
FF /3	CALL *m16:32*	*pm*=56+*m*			Call gate, same privilege
FF /3	CALL *m16:32*	*pm*=90+*m*			Call gate, more privilege, no parameters
FF /3	CALL *m16:32*	*pm*=98+4*x*+*m*			Call gate, more privilege, x parameters
FF /3	CALL *m16:32*	5 + ts			Call to task

* Add one clock for each byte in the next instruction executed (80286 only).

The **CALL** instruction causes the procedure named in the operand to be executed. When the procedure is complete (a return instruction is executed within the procedure), execution continues at the instruction that follows the **CALL** instruction.

The action of the different forms of the instruction are described next.

Near calls are those with destinations of type *r*/*m*16, *r*/*m*32, *rel*16, *rel*32; changing or saving the segment register value is not necessary. The **CALL** *rel*16 and **CALL** *rel*32 forms add a signed offset to the address of the

instruction following **CALL** to determine the destination. The *rel*16 form is used when the instruction's operand-size attribute is 16 bits; *rel*32 is used when the operand-size attribute is 32 bits. The result is stored in the 32-bit EIP register. With *rel* 16, the upper 16 bits of EIP are cleared, resulting in an offset whose value does not exceed 16 bits. **CALL** *r/m*16 and **CALL** *r/m*32 specify a register or memory location from which the absolute segment offset is fetched. The offset fetched from *r/m* is 32 bits for an operand-size attribute of 32 (*r/m*32), or 16 bits for an operand-size of 16 (*r/m*16). The offset of the instruction following **CALL** is pushed onto the stack. It will be popped by a near **RET** instruction within the procedure. The CS register is not changed by this form of **CALL**.

The far calls, **CALL** *ptr*16:16 and **CALL** *ptr*16:32, use a 4-byte or 6-byte operand as a long pointer to the procedure called. The **CALL** *m*16:16 and *m*16:32 forms fetch the long pointer from the memory location specified (indirection). In Real Address Mode or Virtual 8086 Mode, the long pointer provides 16 bits for the CS register and 16 or 32 bits for the EIP register (depending on the operand-size attribute). These forms of the instruction push both CS and IP or EIP as a return address.

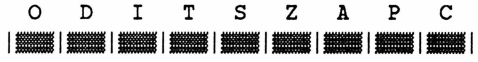
In Protected Mode, both long pointer forms consult the AR byte in the descriptor indexed by the selector part of the long pointer. Depending on the value of the AR byte, the call will perform one of the following types of control transfers:

- a far call to the same protection level
- an inter-protection level far call
- a task switch

Note: Turbo Assembler extends the syntax of the **CALL** instruction to facilitate parameter passing to high-level language routines. See Chapter 5 of the *Turbo Assembler User's Guide* for more details.

CBW

Convert Byte to Word



Opcode	Instruction	Clocks			Description
		386	286	86	
98	CBW	3	2	2	AX sign-extend of AL

CBW converts the signed byte in AL to a signed word in AX by extending the most significant bit of AL (the sign bit) into all of the bits of AH.

CDQ

Convert Doubleword to Quadword

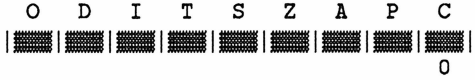
80386 only

Opcode	Instruction	Clocks			Description
		386	286	86	
99	CDQ	2			EDX:EAX ← sign-extend of EAX

CDQ converts the signed doubleword in EAX to a signed 64-bit integer in the register pair EDX:EAX by extending the most significant bit of EAX (the sign bit) into all the bits of EDX.

CLC

Clear Carry Flag

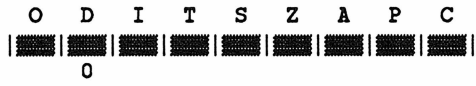


Opcode	Instruction	Clocks		
		386	286	86
F8	CLC	2	2	2

CLC sets the carry flag to zero. It does not affect other flags or registers.

CLD

Clear Direction Flag

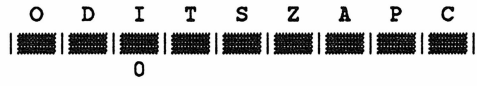


Opcode	Instruction	Clocks			Description
		386	286	86	
FC	CLD	2	2	2	Clear direction flag, SI and DI will increment during string instructions

CLD clears the direction flag. No other flags or registers are affected. After CLD is executed, string operations will increment the index registers (SI and/or DI) that they use.

CLI

Clear Interrupt Flag



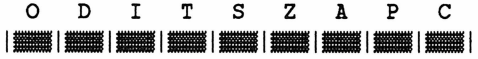
Opcode	Instruction	Clocks		
		386	286	86
FA	CLI	3	3	2

CLI clears the interrupt flag if the current privilege level is at least as privileged as IOPL. No other flags are affected. External interrupts are not recognized at the end of the CLI instruction or from that point on until the interrupt flag is set.

CLTS

Clear Task Switched Flag

80286/80386 protected mode only



TS = 0 (TS is in CR0, not the flag register)

Opcode	Instruction	Clocks		
		386	286	86
0F 06	CLTS	5	2	

CLTS clears the task-switched (TS) flag in register CR0. This flag is set by the 80386 every time a task switch occurs. The TS flag is used to manage processor extensions as follows:

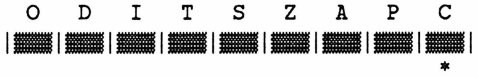
- Every execution of an ESC instruction is trapped if the TS flag if set.
- Execution of a WAIT instruction is trapped if the MP flag and the TS flag are both set.

Thus, if a task switch was made after an ESC instruction was begun, the processor extension's context may need to be saved before a new ESC instruction can be issued. The fault handler saves the context and resets the TS flag.

CLTS appears in operating system software, not in application programs. It is a privileged instruction that can only be executed at privilege level 0.

CMC

Complement Carry Flag

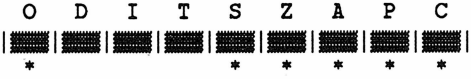


Opcode	Instruction	Clocks		
		386	286	86
F5	CMC	2	2	2

CMC reverses the setting of the carry flag. No other flags are affected.

CMP

Compare Two Operands



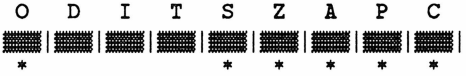
Opcode	Instruction	Clocks			Description
		386	286	86	
3C *ib*	CMP AL,*imm8*	2	3	4	Compare immediate byte to AL
3D *iw*	CMP AX,*imm16*	2	3	4	Compare immediate word from AX
3D *id*	CMP EAX,*imm32*	2			Compare immediate dword to EAX
80 /7 *ib*	CMP *r/m*8,*imm8*	2/5	3/6	4/10+EA	Compare immediate byte to *r/m* byte
81 /7 *iw*	CMP *r/m*16,*imm16*	2/5	3/6	4/10+EA	Compare immediate word to *r/m* word
81 /7 *id*	CMP *r/m*32,*imm32*	2/5			Compare immediate dword to *r/m* dword
83 /7 *ib*	CMP *r/m*16,*imm8*	2/5	3/6	4/10+EA	Compare sign extended immediate byte to *r/m* word
83 /7 *ib*	CMP *r/m*32,*imm8*	2/5			Compare sign extended immediate byte to *r/m* dword
38 /*r*	CMP *r/m*8,r8	2/5	2/7	3/9+EA	Compare byte register to *r/m* byte
39 /*r*	CMP *r/m*16,r16	2/5	2/7	3/9+EA	Compare word register to *r/m* word
39 /*r*	CMP *r/m*32,r32	2/5			Compare dword register to *r/m* dword
3A /*r*	CMP *r*8,*r/m*8	2/6	2/6	3/9+EA	Compare *r/m* byte to byte register
3B /*r*	CMP *r*16,*r/m*8	2/6	2/6	3/9+EA	Compare *r/m* word to word register
3B /*r*	CMP *r*32,*r/m*32	2/6			Compare *r/m* dword to dword register

CMP subtracts the second operand from the first but, unlike the SUB instruction, does not store the result; only the flags are changed. CMP is typically used in conjunction with conditional jumps and the SETcc instruction. If an operand greater than one byte is compared to an immediate byte, the byte value is first sign-extended.

CMPS
CMPSB
CMPSW
CMPSD

Compare String Operands

CMPSD 80386 only



Opcode	Instruction	Clocks			Description
		386	286	86	
A6	CMPS *m8,m8*	10	8	22	Compare bytes ES:[(E)DI] (second operand) with [(E)SI] (first operand)
A7	CMPS *m16,m16*	10	8	22	Compare words ES:[(E)DI] (second operand) with [(E)SI] (first operand)
A7	CMPS*m32,m32*	10			Compare dwords ES:[(E)DI] (second operand) with [(E)SI] (first operand)
A6	CMPSB	10	8	22	Compare bytes ES:[(E)DI] with DS:[SI]
A7	CMPSW	10	8	22	Compare words ES:[(E)DI] with DS:[SI]
A7	CMPSD	10			Compare dwords ES:[(E)DI] with DS:[SI]

CMPS compares the byte, word, or doubleword pointed to by the source-index register with the byte, word, or doubleword pointed to by the destination-index register.

If the address-size attribute of this instruction is 16 bits, SI and DI will be used for source- and destination-index registers; otherwise ESI and EDI will be used. Load the correct index values into SI and DI (or ESI and EDI) before executing CMPS.

The comparison is done by subtracting the operand indexed by the destination-index register from the operand indexed by the source-index register.

Note that the direction of subtraction for CMPS is [SI] – [DI] or [ESI] – [EDI]. The left operand (SI or ESI) is the source and the right operand (DI

or EDI) is the destination. This is the reverse of the usual Intel convention in which the left operand is the destination and the right operand is the source.

The result of the subtraction is not stored; only the flags reflect the change. The types of the operands determine whether bytes, words, or doublewords are compared. For the first operand (SI or ESI), the DS register is used, unless a segment override byte is present. The second operand (DI or EDI) must be addressable from the ES register; no segment override is possible.

After the comparison is made, both the source-index register and destination-index register are automatically advanced. If the direction flag is 0 (CLD was executed), the registers increment; if the direction flag is 1 (STD was executed), the registers decrement. The registers increment or decrement by 1 if a byte is compared, by 2 if a word is compared, or by 4 if a doubleword is compared.

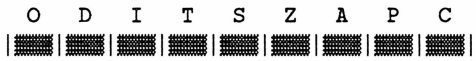
CMPSB, CMPSW and CMPSD are synonyms for the byte, word, and doubleword CMPS instructions, respectively.

CMPS can be preceded by the REPE or REPNE prefix for block comparison of CX or ECX bytes, words, or doublewords. Refer to the description of the REP instruction for more information on this operation.

CWD

Convert Word to Doubleword

80386 only



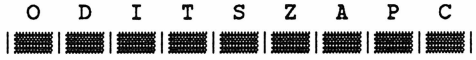
Opcode	Instruction	Clocks			Description
		386	286	86	
99	CWD	2	2	5	DX:AX ← sign-extend of AX

CWD converts the signed word in AX to a signed doubleword in DX:AX by extending the most significant bit of AX into all the bits of DX. Note that CWD is different from CWDE. CWDE uses EAX as a destination, instead of DX:AX.

CWDE

Convert Word to Doubleword

80386 only

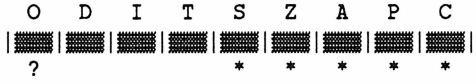


Opcode	Instruction	Clocks			Description
		386	286	86	
98	CWDE	3			EAX sign-extend of AX

CWDE converts the signed word in AX to a doubleword in EAX by extending the most significant bit of AX into the two most significant bytes of EAX. Note that CWDE is different from CWD. CWD uses DX:AX rather than EAX as a destination.

DAA

Decimal Adjust AL After Addition

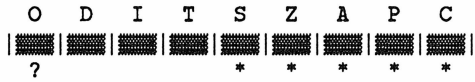


Opcode	Instruction	Clocks		
		386	286	86
27	DAA	4	3	4

Execute DAA only after executing an ADD instruction that leaves a two-BCD-digit byte result in the AL register. The ADD operands should consist of two packed BCD digits. The DAA instruction adjusts AL to contain the correct two-digit packed decimal result.

DAS

Decimal Adjust AL After Subtraction

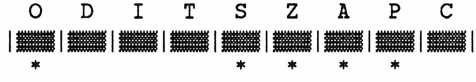


Opcode	Instruction	Clocks			
		386	286	86	
2F	DAS	4	3	4	

Execute DAS only after a subtraction instruction that leaves a two-BCD-digit byte result in the AL register. The operands should consist of two packed BCD digits. DAS adjusts AL to contain the correct packed two-digit decimal result.

DEC

Decrement by 1

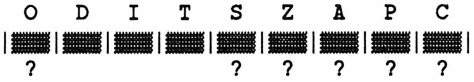


Opcode	Instruction	Clocks			Description
		386	286	86	
FE /1	DEC *r/m*8	2/6	2/7	3/15+EA	Decrement *r/m* byte by 1
FF /1	DEC *r/m*16	2/6	2/7	3/15+EA	Decrement *r/m* word by 1
	DEC *r/m*32	2/6			Decrement *r/m* dword by 1
48+*rw*	DEC *r*16	2	2	3	Decrement word register by 1
48+*rw*	DEC *r*32	2			Decrement dword register by 1

DEC subtracts 1 from the operand. DEC does not change the carry flag. To affect the carry flag, use the SUB instruction with an immediate operand of 1.

DIV

Unsigned Divide



Opcode	Instruction	Clocks			Description
		386	286	86	
F6 /6	DIV AL,*r/m*8	14/17			Unsigned divide AX by *r/m* byte (AL=QUO, AH=REM)
F7 /6	DIV AX,*r/m*16	22/25			Unsigned divide DX:AX by *r/m* word (AX=QUO, DX=REM)
F7 /6	DIV EAX,*r/m*32	38/41			Unsigned divide EDX:EAX by *r/m* dword (EAX=QUO, EDX=REM)

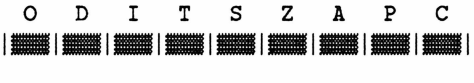
DIV performs an unsigned division. The dividend is implicit; only the divisor is given as an operand. The remainder is always less than the divisor. The type of the divisor determines which registers to use as follows:

Size	Dividend	Divisor	Quotient	Remainder
byte	AX	*r/m*8	AL	AH
word	DX:AX	*r/m*16	AX	DX
dword	EDX:EAX	*r/m*32	EAX	EDX (80386 only)

ENTER

Make Stack Frame for Procedure Parameters

80186/286/386 only



Opcode	Instruction	Clocks			Description
		386	286	86	

C8 *iw* 00	Enter *imm16*,0	10	11	Make procedure stack frame
C8 *iw* 01	Enter *imm16*,1	12	15	Make stack frame for procedure parameters
C8 *iw ib*	Enter *imm16,imm8*	15+4(*n*-1)	12+4(*n*-1)	Make stack frame for procedure parameters

ENTER creates the stack frame required by most block-structured high-level languages. The first operand specifies the number of bytes of dynamic storage allocated on the stack for the routine being entered. The second operand gives the lexical nesting level (0 to 31) of the routine within the high-level language source code. It determines the number of stack frame pointers copied into the new stack frame from the preceding frame. BP (or EBP, if the operand-size attribute is 32 bits) is the current stack frame pointer.

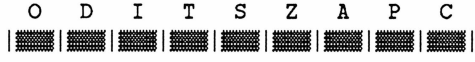
If the operand-size attribute is 16 bits, the processor uses BP as the frame pointer and SP as the stack pointer. If the operand-size attribute is 32 bits, the processor uses EBP for the frame pointer and ESP for the stack pointer.

If the second operand is 0, ENTER pushes the frame pointer (BP or EBP) onto the stack; ENTER then subtracts the first operand from the stack pointer and sets the frame pointer to the current stack-pointer value.

For example, a procedure with 12 bytes of local variables would have an ENTER 12,0 instruction at its entry point and a LEAVE instruction before every RET. The 12 local bytes would be addressed as negative offsets from the frame pointer.

HLT

Halt

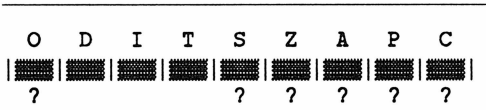


Opcode	Instruction	Clocks		
		386	286	86
F4	HLT	5	2	2

HLT stops instruction execution and places the x86 in a HALT state. An enabled interrupt, NMI, or a reset will resume execution. If an interrupt (including NMI) is used to resume execution after HLT, the saved CS:IP (or CS:EIP on an 80386) value points to the instruction following HLT.

IDIV

Signed Divide



Opcode	Instruction	Clocks			Description
		386	286	86	
F6 /7	IDIV *r/m8*	19	17/20	101-112/ 107-118 +EA	Signed divide AX by *r/m* byte (AL=Quo, AH=Rem)
F7 /7	IDIV AX,*r/m16*	27	25/28	165-184/ 171-190 +EA	Signed divide DX:AX by EA word (AX=Quo, DX=Rem)
F7 /7	IDIV EAX,*r/m32*	43			Signed divide EDX:EAX by DWORD byte (EAX=QUO, EDX=REM)

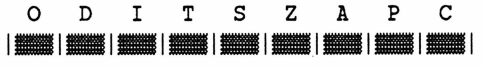
IDIV performs a signed division. The dividend, quotient, and remainder are implicitly allocated to fixed registers. Only the divisor is given as an explicit *r/m* operand. The type of the divisor determines which registers to use as follows:

Size	Divisor	Quotient	Remainder	Dividend
byte	*r/m8*	AL	AH	AX
word	*r/m16*	AX	DX	DX:AX
dword	*r/m32*	EAX	EDX	EDX:EAX (386 only)

If the resulting quotient is too large to fit in the destination, or if the division is 0, an Interrupt 0 is generated. Nonintegral quotients are truncated toward 0. The remainder has the same sign as the dividend and the absolute value of the remainder is always less than the absolute value of the divisor.

IMUL

Signed Multiply



		*		? ? ? ? *	
Opcode	**Instruction**		**Clocks**		**Description**
		386	286	86	
F6 /5	IMUL *r/m8*	9-14/12-17	13/16	80-98/ 86-104+EA	AX ← AL * *r/m* byte
F7 /5	IMUL *r/m16*	9-22/12-25	21/24	128-154/ 134-160+EA	DX:AX ← AX * *r/m* word
F7 /5	IMUL *r/m32*	9-38/12-41			EDX:EAX ← EAX* *r/m* dword
0F AF /r	IMUL *r16,r/m16*	9-22/12-25			word register ← word register * *r/m* word
0F AF /r	IMUL *r32,r/m32*	9-38/12-41			dword register ← dword register * *r/m* dword
6B /r ib	IMUL *r16,r/m16,imm8*	9-14/12-17	21/24		word register ← *r/m*16 * sign-extended immediate byte
6B /r ib	IMUL *r32,r/m32,imm8*	9-14/12-17			dword register ← *r/m*32 * sign-extended immediate byte
6B /r ib	IMUL *r16,imm8*	9-14/12-17	21/24		word register ← word register * sign-extended immediate byte
6B /r ib	IMUL *r32,imm8*	9-14/12-17			dword register ← dword register * sign-extended immediate byte
69 /r iw	IMUL *r16,r/m16,imm16*	9-22/12-25	21/24		word register ← *r/m*16 * immediate word
69 /r id	IMUL r32,*r/m32,imm32*	9-38/12-41			dword register ← *r/m*32 * immediate dword
69 /r iw	IMUL *r16,imm16*	9-22/12-25			word register ← *r/m*16 * immediate word
69 /r id	IMUL r32,*imm32*	9-38/12-41			dword register ← *r/m*32 * immediate dword

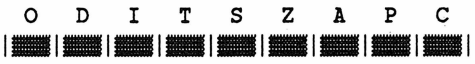
IMUL performs signed multiplication. Some forms of the instruction use implicit register operands. The operand combinations for all forms of the instruction are shown in the "Description" column above.

IMUL clears the overflow and carry flags under the following conditions:

Instruction Form	**Condition for Clearing CF and OF**
r/m8	AL = sign-extend of AL to 16 bits
r/m16	AX = sign-extend of AX to 32 bits
r/m32	EDX:EAX = sign-extend of EAX to 32 bits
r16,r/m16	Result exactly fits within *r16*
r32,r/m32	Result exactly fits within *r32*
r16,r/m16,imm16	Result exactly fits within *r16*
r32,r/m32,imm32	Result exactly fits within *r32*

IN

Input from Port



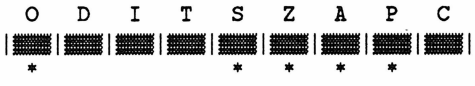
Opcode	Instruction		Clocks		Description
		386	286	86	
E4 *ib*	IN AL,*imm8*	12,pm= 6*/26**	5	10	Input byte from immediate port into AL
E5 *ib*	IN AX,*imm8*	12,pm= 6*/26**	5	10	Input word from immediate port into AX
E5 *ib*	IN EAX,*imm8*	12,pm= 6*/26**			Input dword from immediate port into EAX
EC	IN AL,DX	13,pm= 7*/27**	5	8	Input byte from port DX into AL
ED	IN AX,DX	13,pm= 7*/27**	5	8	Input word from port DX into AX
ED	IN EAX,DX	13,pm= 7*/27**			Input dword from port DX into EAX

*If CPL ≤ IOPL
**If CPL > IOPL or if in virtual 8086 mode

IN transfers a data byte or data word from the port numbered by the second operand into the register (AL, AX, or EAX) specified by the first operand. Access any port from 0 to 65535 by placing the port number in the DX register and using an IN instruction with DX as the second parameter. These I/O instructions can be shortened by using an 8-bit port I/O in the instruction. The upper eight bits of the port address will be 0 when 8-bit port I/O is used.

INC

Increment by 1



Opcode	Instruction	Clocks	Description

		386	286	86	
FE /0	INC *r/m8*	2/6	2/7	3/15+EA	Increment *r/m* byte by 1
FF /0	INC *r/m16*	2/6	2/7	3/15+EA	Increment *r/m* word by 1
FF /6	INC *r/m32*				Increment *r/m* dword by 1
40+ *rw*	INC *r16*	2	2	3	Increment word register by 1
40+ *rd*	INC *r32*				Increment dword register by 1

INC adds 1 to the operand. It does not change the carry flag. To affect the carry flag, use the ADD instruction with a second operand of 1.

INS
INSB
INSW
INSD

Input from Port to String

80186/286/386 only

O	D	I	T	S	Z	A	P	C
▓	▓	▓	▓	▓	▓	▓	▓	▓

Opcode	Instruction	386	286	86	Description
6C	INS *r/m8*,DX	15,pm= 9*/29**	5		Input byte from port DX into ES:(E)DI
6D	INS *r/m16*,DX	15,pm= 9*/29**	5		Input word from port DX into ES:(E)DI
6D	INS *r/m32*,DX	15,pm= 9*/29**			Input dword from port DX into ES:(E)DI
6C	INSB	15,pm= 9*/29**	5		Input byte from port DX into ES:(E)DI
6D	INSW	15,pm= 9*/29**	5		Input word from port DX into ES:(E)DI
6D	INSD	15,pm= 9*/29**			Input dword from port DX into ES:(E)DI

*If CPL ≤ IOPL
**If CPL > IOPL or if in virtual 8086 mode

INS transfers data from the input port numbered by the DX register to the memory byte or word at ES:dest-index. The memory operand must be addressable from ES; no segment override is possible. The destination

register is DI if the address-size attribute of the instruction is 16 bits, or EDI if the address-size attribute is 32 bits.

INS does not allow the specification of the port number as an immediate value. The port must be addressed through the DX register value. Load the correct value into DX before executing the INS instruction.

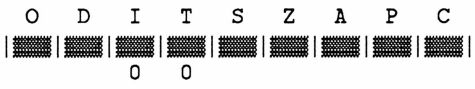
The destination address is determined by the contents of the destination index register. Load the correct index into the destination index register before executing INS.

After the transfer is made, DI or EDI advances automatically. If the direction flag is 0 (CLD was executed), DI or EDI increments; if the direction flag is 1 (STD was executed), DI or EDI decrements. DI increments or decrements by 1 if a byte is input, by 2 if a word is input, or by 4 if a doubleword is input.

INSB, INSW and INSD are synonyms of the byte, word, and doubleword INS instructions. INS can be preceded by the REP prefix for block input of CX bytes or words. Refer to the REP instruction for details of this operation.

INT
INTO

Call to Interrupt Procedure



Opcode	Instruction	Clocks			Description
		386	286	86	
CC	INT3	33	23	52	Interrupt 3—trap to debugger
CC	INT3	pm=59	40		Interrupt 3—Protected Mode, same privilege
CC	INT3	pm=99	78		Interrupt 3—Protected Mode, more privilege
CC	INT3	pm=119			Interrupt 3—from V86 mode to PL0
CC	INT3	ts	167		Interrupt 3—Protected Mode, via task gate
CD *ib*	INT*imm8*	37	23	51	Interrupt numbered by immediate byte
CD *ib*	INT*imm8*	pm=59	40		Interrupt—Protected Mode, same privilege

CD *ib*	INT*imm8*	pm=99	78		Interrupt—Protected Mode, more privilege
CD *ib*	INT*imm8*	pm=119			Interrupt—from V86 mode to PL0
CD *ib*	INT*imm8*	ts	167		Interrupt—Protected Mode, via task gate
CE	INTO	Fail:3, *pm*=3; Pass:35	Fail:3, Pass:24	Fail:4, Pass:53	Interrupt 4—if overflow flag is 1
CE	INTO	pm=59	41		Interrupt 4—Protected Mode, same privilege
CE	INTO	pm=99	79		Interrupt 4—Protected Mode, more privilege
CE	INTO	pm=119			Interrupt 4—from V86 mode to PL0
CE	INTO	ts	168		Interrupt 4—Protected Mode, via task gate

* Add one clock for each byte of the next instruction executed (80286 only).

The INT *n* instruction generates via software a call to an interrupt handler. The immediate operand, from 0 to 255, gives the index number into the Interrupt Descriptor Table (IDT) of the interrupt routine to be called. In Protected Mode, the IDT consists of an array of eight-byte descriptors; the descriptor for the interrupt invoked must indicate an interrupt, trap, or task gate. In Real Address Mode, the IDT is an array of four byte-long pointers. In Protected and Real Address Modes, the base linear address of the IDT is defined by the contents of the IDTR.

The INTO conditional software instruction is identical to the INT *n* interrupt instruction except that the interrupt number is implicitly 4, and the interrupt is made if the 86, 286, or 386 overflow flag is set.

The first 32 interrupts are reserved by Intel for system use. Some of these interrupts are use for internally generated exceptions.

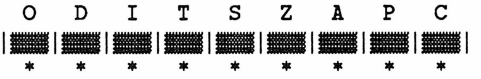
INT *n* generally behaves like a far call except that the flags register is pushed onto the stack before the return address. Interrupt procedures return via the IRET instruction, which pops the flags and return address from the stack.

In Real Address Mode, INT *n* pushes the flags, CS and the return IP onto the stack, in that order, then jumps to the long pointer indexed by the interrupt number.

IRET
IRETD

Interrupt Return

IRETD 80386 only



The flags register is popped from stack.

Opcode	Instruction	Clocks			Description
		386	286	86	
CF	IRET	22,pm=38	17,pm=31	32	Interrupt return (far return and pop flags)
CF	IRET	pm=82	55		Interrupt return, lesser privilege
CF	IRET	ts	169		Interrupt return, different task (NT=1)
CF	IRETD	22,pm=38			Interrupt return (far return and pop flags)
CF	IRETD	pm=82			Interrupt return to lesser privilege
CF	IRETD	pm=60			Interrupt return to V86 mode
CF	IRETD	ts			Interrupt return, different task (NT=1)

* Add one clock, for each byte in the next instruction executed (80286 only).

In Real Address Mode, IRET pops the instruction pointer, CS, and the flags register from the stack and resumes the interrupted routine.

In Protected Mode, the action of IRET depends on the setting of the nested task flag (NT) bit in the flag register. When popping the new flag image from the stack, the IOPL bits in the flag register are changed only when CPL equals 0.

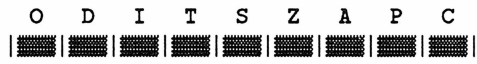
If NT equals 0, IRET returns from an interrupt procedure without a task switch. The code returned to must be equally or less privileged than the interrupt routine (as indicated by the RPL bits of the CS selector popped from the stack). If the destination code is less privileged, IRET also pops the stack pointer and SS from the stack.

If NT equals 1, IRET reverses the operation of a CALL or INT that caused a task switch. The updated state of the task executing IRET is saved in its

task state segment. If the task is reentered later, the code that follows IRET is executed.

Jcc

Jump if Condition is Met



Opcode	Instruction	Clocks			Description
		386	286*	86	
77 *cb*	JA *rel8*	7+*m*,3	7,3	16,4	Jump short if above (CF=0 and ZF=0)
73 *cb*	JAE *rel8*	7+*m*+,3	7,3	16,4	Jump short if above or equal (CF=0)
72 *cb*	JB *rel8*	7+*m*,3	7,3	16,4	Jump short if below (CF=1)
76 *cb*	JBE *rel8*	7+*m*,3	7,3	16,4	Jump short if below or equal (CF=1 or ZF=1)
72 *cb*	JC *rel8*	7+*m*,3	7,3	16,4	Jump short if carry (CF=1)
E3 *cb*	JCXZ *rel8*	9+*m*,5	8,4	18,6	Jump short if CX register is 0
E3 *cb*	JECXZ *rel8*	9+*m*,5			Jump short if ECX register is 0
74 *cb*	JE *rel8*	7+*m*,3	7,3	16,4	Jump short if equal (ZF=1)
74 *cb*	JZ *rel8*	7+*m*,3	7,3	16,4	Jump short if 0 (ZF=1)
7F *cb*	JG *rel8*	7+*m*,3	7,3	16,4	Jump short if greater (ZF=0 and SF=OF)
7D *cb*	JGE *rel8*	7+*m*,3	7,3	16,4	Jump short if greater or equal (SF=OF)
7C *cb*	JL *rel8*	7+*m*,3	7,3	16,4	Jump short if less (SF<>OF)
7E *cb*	JLE *rel8*	7+*m*,3	7,3	16,4	Jump short if less or equal (ZF=1 and SF<>OF
76 *cb*	JNA *rel8*	7+*m*,3	7,3	16,4	Jump short if not above (CF=1 or ZF=1)
72 *cb*	JNAE *rel8*	7+*m*,3	7,3	16,4	Jump short if not above or equal (CF=1)
73 *cb*	JNB *rel8*	7+*m*,3	7,3	16,4	Jump short if not below (CF=0)
77 *cb*	JNBE *rel8*	7+*m*,3	7,3	16,4	Jump short if not below or equal (CF=0 and ZF=0)
73 *cb*	JNC *rel8*	7+*m*,3	7,3	16,4	Jump short if not carry (CF=0)
75 *cb*	JNE *rel8*	7+*m*,3	7,3	16,4	Jump short if not equal (ZF=0)
7E *cb*	JNG *rel8*	7+*m*,3	7,3	16,4	Jump short if not greater (ZF=1 or SF<>OF)
7C *cb*	JNGE *rel8*	7+*m*,3	7,3	16,4	Jump short if not greater or equal (SF<>OF)

7D *cb*	JNL *rel8*	7+*m*,3	7,3	16,4	Jump short if not less (SF=OF)
7F *cb*	JNLE *rel8*	7+*m*,3	7,3	16,4	Jump short if not less or equal (ZF=0 and SF=OF)
71 *cb*	JNO *rel8*	7+*m*,3	7,3	16,4	Jump short if not overflow (OF=0)
7B *cb*	JNP *rel8*	7+*m*,3	7,3	16,4	Jump short if not parity (PF=0)
79 *cb*	JNS *rel8*	7+*m*,3	7,3	16,4	Jump short if not sign (SF=0)
75 *cb*	JNZ *rel8*	7+*m*,3	7,3	16,4	Jump short if not zero (ZF=0)
70 *cb*	JO *rel8*	7+*m*,3	7,3	16,4	Jump short if overflow (OF=1)
7A *cb*	JP *rel8*	7+*m*,3	7,3	16,4	Jump short if parity (PF=1)
7A *cb*	JPE *rel8*	7+*m*,3	7,3	16,4	Jump short if parity even (PF=1)
7B *cb*	JPO *rel8*	7+*m*,3	7,3	16,4	Jump short if parity odd (PF=0)
78 *cb*	JS *rel8*	7+*m*,3	7,3	16,4	Jump short if sign (SF=1)
74 *cb*	JZ *rel8*	7+*m*,3	7,3	16,4	Jump short of zero (ZF=1)
0F 87 *cw/cd*	JA *rel16/32*	7+*m*,3			Jump near if above (CF=0 and ZF=0)
0F 83 *cw/cd*	JAE *rel16/32*	7+*m*,3			Jump near if above or equal (CF=0)
0F 82 *cw/cd*	JB *rel16/32*	7+*m*,3			Jump near if below (CF=1)
0F 86 *cw/cd*	JBE *rel16/32*	7+*m*,3			Jump near if below or equal (CF=1or ZF=1)
0F 82 *cw/cd*	JC *rel16/32*	7+*m*,3			Jump near if carry (CF=1)
0F 84 *cw/cd*	JE *rel16/32*	7+*m*,3			Jump near if equal (ZF=1)
0F 84 *cw/cd*	JZ *rel16/32*	7+*m*,3			Jump near if 0 (ZF=1)
0F 8F *cw/cd*	JG *rel16/32*	7+*m*,3			Jump near if greater (ZF=0 and SF=OF)
0F 8D *cw/cd*	JGE *rel16/32*	7+*m*,3			Jump near if greater or equal (SF=OF)
0F 8C *cw/cd*	JL *rel16/32*	7+*m*,3			Jump near if less (SF<>OF)
0F 8E *cw/cd*	JLE *rel16/32*	7+*m*,3			Jump near if less or equal (ZF=1 and SF<>OF)
0F 86 *cw/cd*	JNA *rel16/32*	7+*m*,3			Jump near if not above (CF=1 or ZF=1)
0F 82 *cw/cd*	JNAE *rel16/32*	7+*m*,3			Jump near if not above or equal (CF=1)
0F 83 *cw/cd*	JNB *rel16/32*	7+*m*,3			Jump near if not below (CF=0)
0F 87 *cw/cd*	JNBE *rel16/32*	7+*m*,3			Jump near if not below or equal (CF=0 and ZF=0
0F 83 *cw/cd*	JNC *rel16/32*	7+*m*,3			Jump near if not carry and ZF=0)
0F 85 *cw/cd*	JNE *rel16/32*	7+*m*,3			Jump near if not equal (ZF=0)
0F 8E	JNG *rel16/32*	7+*m*,3			Jump near if not greater

cw/cd			(ZF=1 or SF<>OF)
0F 8C	JNGE *rel16/32*	7+*m*,3	Jump near if not greater or
cw/cd			equal (SF<>OF)
0F 8D	JNL *rel16/32*	7+*m*,3	Jump near if not less
cw/cd			(SF=OF)
0F 8F	JNLE *rel16/32*	7+*m*,3	Jump near if not less or
cw/cd			equal (ZF=0 and SF=OF)
0F 81	JNO *rel16/32*	7+*m*,3	Jump near if not overflow
cw/cd			(OF=0)
0F 8B	JNP *rel16/32*	7+*m*,3	Jump near if not parity
cw/cd			(PF=0)
0F 89	JNS *rel16/32*	7+*m*,3	Jump near if not sign
cw/cd			(SF=0)
0F 85	JNZ *rel16/32*	7+*m*,3	Jump near if not zero
cw/cd			(ZF=0)
0F 80	JO *rel16/32*	7+*m*,3	Jump near if overflow
cw/cd			(OF=1)
0F 8A	JP *rel16/32*	7+*m*,3	Jump near if parity (PF=1)
cw/cd			
0F 8A	JPE *rel16/32*	7+*m*,3	Jump near if parity even
cw/cd			(PF=1)
0F 8B	JPO *rel16/32*	7+*m*,3	Jump near if parity odd
cw/cd			(PF=0)
0F 88	JS *rel16/32*	7+*m*,3	Jump near if sign (SF=1)
cw/cd			
0F 84	JZ *rel16/32*	7+*m*,3	Jump near if zero (ZF=1)
cw/cd			

* When a jump is taken, add one clock for every byte of the next instruction executed (80286 only).

Notes: The first clock count is for the true condition (branch taken); the second clock count is for the false condition (branch not taken). *rel16/32* indicates that these instructions map to two; one with a 16-bit relative displacement, the other with a 32-bit relative displacement, depending on the operanc-size attribute of the instruction.

Conditional jumps (except JCXZ/JECXZ) test the flags which have been set by a previous instruction. The conditions for each mnemonic are given in parentheses after each description above. The terms "less" and "greater" are used for comparisons of signed integers; "above" and "below" are used for unsigned integers.

If the given condition is true, a jump is made to the location provided as the operand. Instruction coding is most efficient when the target for the conditional jump is in the current code segment and within –128 to + 127 bytes of the next instruction's first byte. The jump can also target –32768 thru +32767 (segment size attribute 16) or –2 to the 31st power +2 to the 31st power –1 (segment size attribute 32) relative to the next instruction's first byte. When the target for the conditional jump is in a different segment, use the opposite case of the jump instruction (i.e., JE and JNE), and then access the target with an unconditional far jump to the other segment. For example, you cannot code

```
JZ FARLABEL;
```

You must instead code

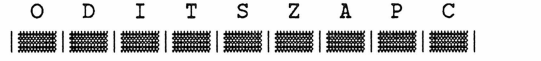
```
        JNZ BEYOND;
        JMP FARLABEL;
BEYOND:
```

Because there can be several ways to interpret a particular state of the flags, TASM provides more than one mnemonic for most of the conditional jump opcodes. For example, if you compared two characters in AX and want to jump if they are equal, use JE; or, if you ANDed AX with a bit field mask and only want to jump if the result is 0, use JZ, a synonym for JE.

JCXZ/JECXZ differs from other conditional jumps because it tests the contents of the CX or ECX register for 0, not the flags. JCXZ/JECXZ is useful at the beginning of a conditional loop that terminates with a conditional loop instruction (such as LOOPNE TARGET LABEL). The JCXZ/JECXZ prevents entering the loop with CX or ECX equal to zero, which would cause the loop to execute 64K or 32G times instead of zero times.

JMP

Jump



All if a task switch takes place; none if no task switch occurs.

Opcode	Instruction	Clocks			Description
		386	286	86	
EB *cb*	JMP *rel8*	7+*m*	7	15	Jump short
E9 *cw*	JMP *rel16*	7+*m*	7	15	Jump near, displacement to next instruction
FF */4*	JMP *r/m16*	7+*m*/10+*m*	7/11	11/18+EA	Jump near indirect
EA *cd*	JMP *ptr16:16*	12+*m*,*pm*= 27+*m*	11,*pm*= 23	15	Jump intersegment, 4-byte immediate address
EA *cd*	JMP *ptr16:16*	*pm*=45+*m*	38		Jump to call gate, same privilege
EA *cd*	JMP *ptr16:16*	ts	175		Jump via task state segment
EA *cd*	JMP *ptr16:16*	ts	180	24+EA	Jump via task gate
FF */5*	JMP *m16:16*	43+*m*, *pm*=31+*m*	15,*pm*=26		Jump *r/m16:16* indirect and intersegment
FF */5*	JMP *m16:16*	*pm*=49+*m*	41		Jump to call gate, same privilege
FF */5*	JMP *m16:16*	5+ts	178		Jump via task state segment
FF */5*	JMP *m16:16*	5+ts	183		Jump via task gate

E9 *cd*	JMP *rel32*	7+*m*	Jump near, displacement relative to next instruction
FF /4	JMP *r/m32*	7+*m*,10+*m*	Jump near, indirect
EA *cp*	JMP *ptr16:32*	12+*m*,*pm*= 27+*m*	Jump intersegment, 6-byte immediate address
EA *cp*	JMP *ptr16:32*	*pm*=45+*m*	Jump to call gate, same privilege
EA *cp*	JMP *ptr16:32*	ts	Jump via task state segment
EA *cp*	JMP *ptr16:32*	ts	Jump via task gate
FF /5	JMP *m16:32*	43+*m*,*pm*= 31+*m*	Jump intersegment, address at *r/m* dword
FF /5	JMP *m16:32*	*pm*=49+*m*	Jump to call gate, same privilege
FF /5	JMP *m16:32*	5 + ts	Jump via task state segment
FF /5	JMP *m16:32*	5 + ts	Jump via task gate

* Add one clock for every byte of the next instruction executed (80286 only).

The JMP instruction transfers control to a different point in the instruction stream without recording return information.

The action of the various forms of the instruction are shown below.

Jumps with destinations of type *r/m16*, *r/m32*, *rel16*, and *rel32* are near jumps and do not involve changing the segment register value.

The JMP *rel16* and JMP *rel32* forms of the instruction add an offset to the address of the instruction following the JMP to determine the destination. The *rel16* form is used when the instruction's operand-size attribute is 16 bits (segment size attribute 16 only); *rel32* is used when the operand-size attribute is 32 bits (segment size attribute 32 only). The result is stored in the 32-bit EIP register. With *rel16*, the upper 16 bits of EIP are cleared, which results in an offset whose value does not exceed 16 bits.

JMP *r/m16* and JMP *r/m32* specifies a register or memory location from which the absolute offset from the procedure is fetched. The offset fetched from *r/m* is 32 bits for an operand-size attribute of 32 bits (*r/m32*), or 16 bits for an operand-size attribute of 16 bits (*r/m16*).

The JMP *ptr16:16* and *ptr16:32* forms of the instruction use a four-byte or six-byte operand as a long pointer to the destination. The JMP *m16:16* and *m16:32* forms fetch the long pointer from the memory location specified (indirection). In Real Address Mode or Virtual 8086 Mode, the long pointer provides 16 bits for the CS register and 16 or 32 bits for the EIP register (depending on the operand-size attribute). In Protected Mode, both long pointer forms consult the Access Rights (AR) byte in the descriptor indexed by the selector part of the long pointer. Depending on the value of the AR byte, the jump will perform one of the following types of control transfers:

- a jump to a code segment at the same privilege level
- a task switch

LAHF

Loads Flags into AH Register

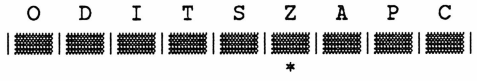
Opcode	Instruction	Clocks			Description
		386	286	86	
9F	LAHF	2	2	4	Load: AH = flags SF ZF xx AF xx PF xx CF

LAHF transfers the low byte of the flags word to AH. The bits, from MSB to LSB, are sign, zero, indeterminate, auxiliary carry, indeterminate, parity, indeterminate, and carry.

LAR

Load Access Rights Byte

80286/386 protected mode only



Opcode	Instruction	Clocks			Description
		386	286	86	
0F 02 /r	LAR *r16,r/m16*	pm=15/16	14/16		*r16* ← *r/m16* masked by FF00
0F 02 /r	LAR *r32,r/m32*	pm=15/16			*r32* ← *r/m32* masked by 00FxFF00

The LAR instruction stores a marked form of the second doubleword of the descriptor for the source selector if the selector is visible at the CPL (modified by the selector's RPL) and is a valid descriptor type. The destination register is loaded with the high-order doubleword of the descriptor masked by 00FxFF00, and ZF is set to 1. The *x* indicates that the four bits corresponding to the upper four bits of the limit are

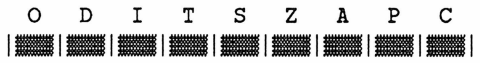
undefined in the value loaded by LAR. If the selector is invisible or of the wrong type, ZF is cleared.

If the 32-bit operand size is specified, the entire 32-bit value is loaded into the 32-bit destination register. If the 16-bit operand size is specified, the lower 16-bits of this value are stored in the 16-bit destination register.

All code and data segment descriptors are valid for LAR. (See your Intel manual for valid segment and gate descriptor types for LAR.)

LEA

Load Effective Address Offset



Opcode	Instruction	Clocks			Description
		386	286	86	
8D/*r*	LEA *r16,m*	2	3	2+EA	Store effective address for *m* in register *r16*
8D/*r*	LEA *r32,m*	2			Store effective address for *m* in register *r32*
8D/*r*	LEA *r16,m*	2			Store effective address for *m* in register *r16*
8D/*r*	LEA *r32,m*	2			Store effective address for *m* in register *r32*

LEA calculates the effective address (offset part) and stores it in the specified register. The operand-size attribute of the instruction is determined by the chosen register. The address-size attribute is determined by the USE attribute of the segment containing the second operand. The address-size and operand-size attributes affect the action performed by LEA, as follows:

Operand Size	Address Size	Action Performed
16	16	16-bit effective address is calculated and stored in requested 16-bit register destination.
16	32	32-bit effective address is calculated. The lower 16 bits of the address are stored in the requested 16-bit register destination.
32	16	16-bit effective address is calculated. The 16-bit address is zero-extended and stored in the requested 32-bit register destination.

32	32	32-bit effective address is calculated and stored in the requested 32-bit register destination.

LEAVE

High Level Procedure Exit

80186/286/386 only

Opcode	Instruction	Clocks			Description
		386	286	86	
C9	LEAVE	4	5		Set SP to BP, then pop BP
C9	LEAVE	4			Set ESP to EBP, then pop EBP

LEAVE reverses the actions of the ENTER instruction. By copying the frame pointer to the stack pointer, LEAVE releases the stack space used by a procedure for its local variables. The old frame pointer is popped into BP or EBP, restoring the caller's frame. A subsequent RET *nn* instruction removes any arguments pushed onto the stack of the exiting procedure.

LGDT/LIDT

Load Global/Interrupt Descriptor Table Register

80286/386 protected mode only

Opcode	Instruction	Clocks			Description
		386	286	86	
0F 01 /2	LGDT *m16&32*	11	11		Load *m* into Global Descriptor Table reg
0F 01 /3	LIDT *m16&32*	11	12		Load *m* into Interrupt Descriptor Table reg

The LGDT and LIDT instructions load a linear base address and limit value from a six-byte data operand in memory into the GDTR or IDTR, respectively. If a 16-bit operand is used with LGDT or LIDT, the register is loaded with a 16-bit limit and a 24-bit base, and the high-order eight bits of the six-byte data operand are not used. If a 32-bit operand is used, a 16-bit limit and a 32-bit base is loaded; the high-order eight bits of the six-byte operand are used as high-order base address bits.

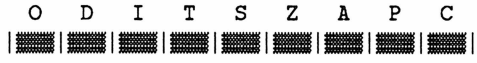
The SGDT and SIDT instructions always store into all 48 bits of the six-byte data operand. With the 80286, the upper eight bits are undefined after SGDT or SIDT is executed. With the 80386, the upper eight bits are written with the high-order eight address bits, for both a 16-bit operand and a 32-bit operand. If LGDT or LIDT is used with a 16-bit operand to load the register stored by SGDT or SIDT, the upper eight bits are stored as zeros.

LGDT and LIDT appear in operating system software; they are not used in application programs. They are the only instructions that directly load a linear address (i.e., not a segment relative address) in 80386 Protected Mode.

LGS
LSS
LFS
LDS
LES

Load Full Pointer

LGS/LSS/LFS 80386 only



Opcode	Instruction		Clocks		Description
		386	286	86	
C5 /r	LDS *r16,m16:16*	7,*pm*=22	7,*pm*=21	16+EA	Load DS:*r16* with pointer from memory
C5 /r	LDS *r32,m16:32*	7,*pm*=22			Load DS:*r32* with pointer from memory
0F B2 /r	LSS *r16,m16:16*	7,*pm*=22			Load SS:*r16* with pointer from memory
0F B2 /r	LSS *r32,m16:32*	7,*pm*=22			Load SS:*r32* with pointer from memory

C4 /r	LES *r16,m16:16*	7,*pm*=22	7,*pm*=21	16+EA	Load ES:*r16* with pointer from memory
C4 /r	LES *r32,m16:32*	7,*pm*=22			Load ES:*r32* with pointer from memory
0F B4 /r	LFS *r16,m16:16*	7,*pm*=25			Load FS:*r16* with pointer from memory
0F B4 /r	LFS *r32,m16:32*	7,*pm*=25			Load FS:*r32* with pointer from memory
0F B5 /r	LGS *r16,m16:16*	7,*pm*=25			Load GS:*r16* with pointer from memory
0F B5 /r	LGS *r32,m16:32*	7,*pm*=25			Load GS:*r32* with pointer from memory

These instructions read a full pointer from memory and store it in the selected segment register: register pair. The full pointer loads 16 bits into the segment register SS, DS, ES, FS, or GS. The other register loads 32 bits if the operand-size attribute is 32 bits, or loads 16 bits if the operand-size attribute is 16 bits. The other 16- or 32-bit register to be loaded is determined by the *r*16 or *r*32 register operand specified.

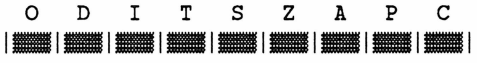
When an assignment is made to one of the segment registers, the descriptor is also loaded into the segment register. The data for the register is obtained from the descriptor table entry for the selector given.

A null selector (values 0000-0003) can be loaded into DS, ES, FS, or GS registers without causing a protection exception. (Any subsequent reference to a segment whose corresponding segment register is loaded with a null selector to address memory causes a #GP(0) exception. No memory reference to the segment occurs.)

LLDT

Load Local Descriptor Table Register

80286/386 protected mode only



Opcode	Instruction	Clocks			Description
		386	286	86	
0F 00 /2	LLDT *r/m* 16	20	17/19		Load selector *r/m*16 into LDTR

LLDT loads the Local Descriptor Table register (LDTR). The word operand (memory or register) to LLDT should contain a selector to the

Global Descriptor Table (GDT). The GDT entry should be a Local Descriptor Table. If so, then the LDTR is loaded from the entry. The descriptor registers DS, ES, SS, FS, GS, and CS are not affected. The LDT field in the task state segment does not change.

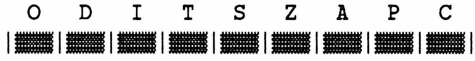
The selector operand can be 0; if so, the LDTR is marked invalid. All descriptor references (except by the LAR, VERR, VERW or LSL instructions) cause a #GP fault.

LLDT is used in operating system software; it is not used in application programs.

LMSW

Load Machine Status Word

80286/386 protected mode only



Opcode	Instruction	Clocks			Description
		386	286	86	
0F 01 /6	LMSW *r/m* 16	10/13	3/6		Load *r/m* 16 into Machine Status Word

LMSW loads the machine status word (part of CR0) from the source operand. This instruction can be used to switch to Protected Mode; if so, it must be followed by an intrasegment jump to flush the instruction queue. LMSW will not switch back to Real Address Mode.

LMSW is used only in operating system software. It is not used in application programs.

LOCK

Assert LOCK# Signal Prefix

Opcode	Instruction	Clocks	Description

		386	286	86	
F0	LOCK	0	0	2	Assert LOCK# signal for the next instruction

The LOCK prefix causes the LOCK# signal of the CPU to be asserted during execution of the instruction that follows it. In a multiprocessor environment, this signal can be used to ensure that the CPU has exclusive use of any shared memory while LOCK# is asserted. The read-modify-write sequence typically used to implement test-and-set on the 80386 is the BTS instruction.

The LOCK prefix functions only with the following instructions:

BT, BTS, BTR, BTC	mem, reg/imm
XCHG	reg, mem
XCHG	mem, reg
ADD, OR, ADC, SBB, AND, SUB, XOR	mem, reg/imm
NOT, NEG, INC, DEC	mem

An undefined opcode trap will be generated if a LOCK prefix is used with any instruction not listed above.

XCHG always asserts LOCK # regardless of the presence or absence of the LOCK prefix.

The integrity of the LOCK is not affected by the alignment of the memory field. Memory locking is observed for arbitrarily misaligned fields.

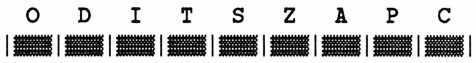
Locked access is *not* assured if another CPU processor is executing an instruction concurrently that has one of the following characteristics:

- Is not preceded by a LOCK prefix.
- Is not one of the instructions in the preceding list.
- Specifies a memory operand that does not exactly overlap the destination operand. Locking is not guaranteed for partial overlap, even if one memory operand is wholly contained within another.

LODS
LODSB
LODSW
LODSD

Load String Operand

LODSD 80386 only



Opcode	Instruction	Clocks			Description
		386	286	86	
AC	LODS *m18*	5	5	12	Load byte [(E)SI] into AL
AD	LODS *m16*	5	5	12	Load word [(E)SI] into AX
AD	LODS *m32*	5			Load dword [(E)SI] into EAX
AC	LODSB	5	5	12	Load byte DS:[(E)SI] into AL
AD	LODSW	5	5	12	Load word DS:[(E)SI] into AX
AD	LODSD	5			Load dword DS:[(E)SI] into EAX

LODS loads the AL, AX, or EAX register with the memory byte, word, or doubleword at the location pointed to by the source-index register. After the transfer is made, the source-index register is automatically advanced. If the direction flag is 0 (CLD was executed), the source index increments; if the direction flag is 1 (STD was executed), it decrements. The increment or decrement is 1 if a byte is loaded, 2 if a word is loaded, or 4 if a doubleword is loaded.

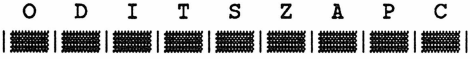
If the address-size attribute for this instruction is 16 bits, SI is used for the source-index register; otherwise the address-size attribute is 32 bits, and the ESI register is used. The address of the source data is determined solely by the contents of ESI/SI. Load the correct index value into SI before executing the LODS instruction. LODSB, LODSW, LODSD are synonyms for the byte, word, and doubleword LODS instructions.

LODS can be preceded by the REP prefix; however, LODS is used more typically within a LOOP construct, because further processing of the data moved into EAX, AX, or AL is usually necessary.

LOOP
LOOPcond

Loop Control with CX Counter

Loop Control with CX/ECX Counter 80386 only



Opcode	Instruction	Clocks			Description
		386	286	86	
E2 *cb*	LOOP *rel8*	11+m	8,noj=4	17,noj=5	DEC Count; jump short if Count <> 0
E1 *cb*	LOOPE *rel8*	11+m	8,noj=4	18,noj=6	DEC Count; jump short if Count <> 0 and ZF=1
E1 *cb*	LOOPZ *rel8*	11+m	8,noj=4	18,noj=6	DEC Count; jump short if Count <> 0 and ZF=1
E0 *cb*	LOOPNE *rel8*	11+m	8,noj=4	19,noj=5	DEC Count; jump short if Count <> 0 and ZF=0
E0 *cb*	LOOPNZ *rel8*	11+m	8,noj=4	19,noj=5	DEC Count; jump short if Count <> 0 and ZF=0

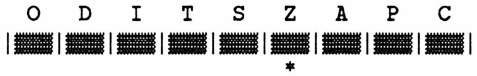
LOOP decrements the count register without changing any of the flags. Conditions are then checked for the form of LOOP being used. If the conditions are met, a short jump is made to the label given by the operand to LOOP. If the address-size attribute is 16 bits, the CX register is used as the count register; otherwise the ECX register is used (80386 only). The operand of LOOP must be in the range from 128 (decimal) bytes before the instruction to 127 bytes ahead of the instruction.

The LOOP instructions provide iteration control and combine loop index management with conditional branching. Use the LOOP instruction by loading an unsigned iteration count into the count register, then code the LOOP at the end of a series of instructions to be iterated. The destination of LOOP is a label that points to the beginning of the iteration.

LSL

Load Segment Limit

80286/386 protected mode only



Opcode	Instruction	Clocks			Description
		386	286	86	
0F 03 /r	LSL *r16,r/m16*	*pm*=20/21	14/16		Load: *r16* ← segment limit, selector *r/m*16 (byte granular)
0F 03 /r	LSL *r32,r/m32*	*pm*=20/21			Load: *r32* ← segment limit, selector *r/m*32 (byte granular)
0F 03 /r	LSL *r16,r/m16*	*pm*=25/26	14/16		Load: *r16* ← segment limit, selector *r/m*16 (page granular)
0F 03 /r	LSL *r32,r/m32*	*pm*=26/26			Load: *r32* ← segement limit selector *r/m*32 (page granular)

The LSL instruction loads a register with an unscrambled segment limit, and sets ZF to 1, provided that the source selector is visible at the CPL weakened by RPL, and that the descriptor is a type accepted by LSL. Otherwise, ZF is cleared to 0, and the destination register is unchanged. The segment limit is loaded as a byte granular value. If the descriptor has a page granular segment limit, LSL will translate it to a byte limit before loading it in the destination register (shift left 12 the 20-bit "raw" limit from descriptor, then OR with 00000FFFH).

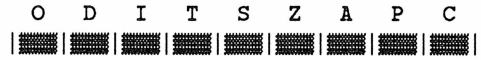
The 32-bit forms of this instruction store the 32-bit byte granular limit in the 16-bit destination register.

Code and data segment descriptors are valid for LSL.

LTR

Load Task Register

80286/386 protected mode only



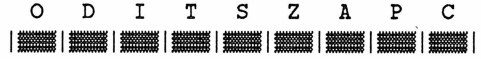
Opcode	Instruction	Clocks			Description
		386	286	86	
0F 00 /3	LTR *r/m16*	pm=23/27	17/19		Load EA word into task register

LTR loads the task register from the source register or memory location specified by the operand. The loaded task state segment is marked busy. A task switch does not occur.

LTR is used only in operating system software; it is not used in application programs.

MOV

Move Data



Opcode	Instruction	Clocks			Description
		386	286	86	
88 /r	MOV *r/m8,r8*	2/2	2/3	2/9+EA	Move byte register into *r/m* byte
89 /r	MOV *r/m16,r16*	2/2	2/3	2/9+EA	Move word register into *r/m* word
89 /r	MOV *r/m32,r32*	2/2			Move dword register to *r/m* dword
8A /r	MOV *r8,r/m8*	2/4	2/5	2/8+EA	Move *r/m* byte into byte register
8B /r	MOV *r16,r/m16*	2/4	2/5	2/8+EA	Move *r/m* word into word register
8B /r	MOV *r32,r/m32*	2/4			Move *r/m* dword into dword register

8C */r*	MOV *r/m16,Sreg*	2/2	2/3	2/9+EA	Move segment register to *r/m* register
8D */r*	MOV *Sreg,r/m16*	2/5,*pm*= 18/19	2/5,*pm*= 17/19	2/8+EA	Move *r/m* word to segment register
A0	MOV AL,*moffs8*	4	5	10	Move byte at (seg:offset) to AX
A1	MOV AX,*moffs16*	4	5	10	Move word at (seg:offset) to AX
A1	MOV EAX,*moffs32*	4			Move dword at (seg:offset) to EAX
A2	MOV *moffs8*,AL	4	3	10	Move AL to (seg:offset)
A3	MOV *moffs16*,AX	2	3	10	Move AX to (seg:offset)
A3	MOV *moffs32*, EAX	2			Move EAX to (seg:offset)
B0+ rb	MOV *reg8,imm8*	2	2	4	Move immediate byte to register
B8+ rw	MOV *reg16,imm16*	2	2	4	Move immediate word to register
B8+rd	MOV *reg32,imm32*	2			Move immediate dword to register
C6	MOV *r/m8,imm8*	2/2	2/3	4/10+EA	Move immediate byte to *r/m* byte
C7	MOV *r/m16,imm16*	2/2	2/3	4/10+EA	Move immediate word to *r/m* word
C7	MOV *r/m32,imm32*	2/2			Move immediate dword to *r/m* dword

MOV copies the second operand to the first operand.

If the destination operand is a segment register (DS, ES, SS, etc.), then data from a descriptor is also loaded into the register. The data for the register is obtained from the descriptor table entry for the selector given. A null selector (values 0000-0003) can be loaded into DS and ES registers without causing an exception; however, use of DS or ES causes a #GP(0), and no memory reference occurs.

A MOV into SS inhibits all interrupts until after the execution of the next instruction (which is presumably a MOV into eSP).

MOV

Move to/from Special Registers

80386 only

O	D	I	T	S	Z	A	P	C

Opcode	Instruction	Clocks	Description

		386	
0F 20 /r	MOV *r32,CR0/CR2/CR3*	6	Move (control register) to (register)
0F 22 /r	MOV *CR0/CR2/CR3,r32*	10/4/5	Move (register) to (control register)
0F 21 /r	MOV *r32,DR0 – 3*	22	Move (debug register) to (register)
0F 21 /r	MOV *r32,DR6/DR7*	14	Move (debug register) to (register)
0F 23 /r	MOV *DR0 –3,r32*	22	Move (register) to (debug register)
0F 23 /r	MOV *DR6/DR7,r32*	16	Move (register) to (debug register)
0F 24 /r	MOV *r32,TR6/TR7*	12	Move (test register) to (register)
0F 26 /r	MOV *TR6/TR7,r32*	12	Move (register) to (test register)

These forms of MOV store or load the following special registers in or from a general-purpose register:

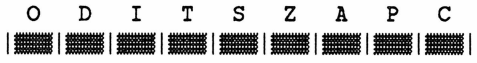
- Control registers CRO, CR2, and CR3
- Debug Registers DRO, DR1, DR2, DR3, DR6, and DR7
- Test Registers TR6 and TR7

32-bit operands are always used with these instructions, regardless of the operand-size attribute.

MOVS
MOVSB
MOVSW
MOVSD

Move Data from String to String

MOVSD 80386 only



Opcode	Instruction	Clocks			Description
		386	286	86	
A4	MOVS *m8,m8*	7	5	18	Move byte [(E)SI] to ES:[(E)DI]
A5	MOVS *m16,m16*	7	5	18	Move word [(E)SI] to ES:[(E)DI]
A5	MOV*m32,m32*	7			Move dword [(E)SI] to ES:[(E)DI]
A4	MOVSB	7	5	18	Move byte DS:[(E)SI] to ES:[(E)DI]
A5	MOVSW	7	5	18	Move word DS:[(E)SI] to ES:[(E)DI]

A5	MOVSD	7	Move dword DS:[(E)SI] to ES:[(E)DI]

MOVS copies the byte or word at [(E)SI] to the byte or word at ES:[(E)DI]. The destination operand must be addressable from the ES register; no segment override is possible for the destination. A segment override can be used for the source operand; the default is DS.

The addresses of the source and destination are determined solely by the contents of (E)SI and (E)DI. Load the correct index values into (E)SI and (E)DI before executing the MOVS instruction. MOVSB, MOVSW, and MOVSD are synonyms for the byte, word, and doubleword MOVS instructions.

After the data is moved, both (E)SI and (E)DI are advanced automatically. If the direction flag is 0 (CLD was executed), the registers are incremented; if the direction flag is 1 (STD was executed), the registers are decremented. The registers are incremented or decremented by 1 if a byte was moved, 2 if a word was moved, or 4 if a doubleword was moved.

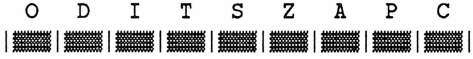
MOVS can be preceded by the REP prefix for block movement of CX bytes or words. Refer to the REP instruction for details of this operation.

MOVSX

Move with Sign-Extend

80386 only

O D I T S Z A P C



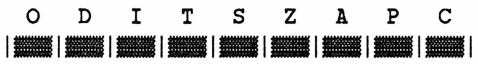
Opcode	Instruction	Clocks			Description
		386	286	86	
0F BE /r	MOVSX *r16,r/m8*	3/6			Move byte to word with sign-extend
0F BE /r	MOVSX *r32,r/m8*	3/6			Move byte to dword, sign extend
0F BE /r	MOVSX *r32,r/m16*	3/6			Move word to dword,

MOVSX reads the contents of the effective address or register as a byte or a word, sign-extends the value to the operand-size attribute of the instruction (16 or 32 bits), and stores the result in the destination register.

MOVZX

Move with Zero-Extend

80386 only

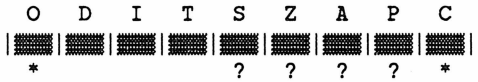


Opcode	Instruction	Clocks			Description
		386	286	86	
0F B6 /r	MOVZX *r16,r/m8*	3/6			Move byte to word with zero-extend
0F B6 /r	MOVZX *r32,r/m8*	3/6			Move byte to dword, zero-extend
0F B7 /r	MOVZX *r32,r/m16*	3/6			Move word to dword, zero-extend

MOVZX reads the contents of the effective address or register as a byte or a word, zero extends the value to the operand-size attribute of the instruction (16 or 32 bits), and stores the result in the destination register.

MUL

Unsigned Multiplication of AL or AX



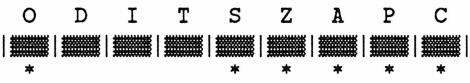
Opcode	Instruction	Clocks			Description
		386	286	86	
F6 /4	MUL AL,*r/m8*	9-14/12-17	13/16	70-77/ 76-83+EA	Unsigned multiply (AX ← AL 8 *r/m* byte)
F7 /4	MUL AX,*r/m16*	9-22/12-25	21/24	118-113/ 124-139 +EA	Unsigned multiply (DX:AX ← AX * *r/m* word)
F7 /4	MUL EAX,*r/m32*	9-38/12-41			Unsigned multiply (EDX:EAX ← EAX * *r/m* dword)

MUL performs unsigned multiplication. Its actions depend on the size of its operand, as follows:

- A byte operand is multiplied by AL; the result is left in AX. The carry and overflow flags are set to 0 if AH is 0; otherwise, they are set to 1.
- A word operand is multiplied by AX; the result is left in DX: AX. DX contains the high-order 16 bits of the product. The carry and overflow flags are set to 0 if DX is 0; otherwise, they are set to 1.
- A doubleword operand is multiplied by EAX and the result is left in EDX:EAX. EDX contains the high-order 32 bits of the product. The carry and overflow flags are set to 0 if EDX is 0; otherwise, they are set to 1 (80386 only).

NEG

Two's Complement Negation



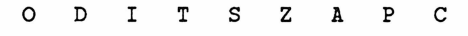
Opcode	Instruction	Clocks			Description
		386	286	86	
F6 /3	NEG *r/m8*	2/6	2/7	3/16+EA	Two's complement negate *r/m* byte
F7 /3	NEG *r/m16*	2/6	2/7	3/16+EA	Two's complement negate *r/m* word
F7 /3	NEG *r/m32*	2/6			Two's complement negate *r/m* dword

NEG replaces the value of a register or memory operand with its two's complement. The operand is subtracted from zero, and the result is placed in the operand.

The carry flag is set to 1, unless the operand is zero, in which case the carry flag is cleared to 0.

NOP

No OPERATION



Opcode	Instruction	Clocks		
		386	286	86
90	NOP	3	3	3

NOP performs no operation. NOP is a one-byte instruction that takes up space but affects none of the machine context except (E)IP.

NOP is an alias mnemonic for the XCHG (E)AX, (E)AX instruction.

NOT

One's Complement Negation

O	D	I	T	S	Z	A	P	C

Opcode	Instruction	Clocks			Description
		386	286	86	
F6 /2	NOT *r/m8*	2/6	2/7	3/16+EA	Reverse each bit of *r/m* byte
F7 /2	NOT *r/m16*	2/6	2/7	3/16+EA	Reverse each bit of *r/m* word
F7 /2	NOT *r/m32*	2/6	2/7		Reverse each bit of *r/m* dword

NOT inverts the operand; every 1 becomes a 0, and vice versa.

OR

Logical Inclusive OR

O	D	I	T	S	Z	A	P	C
0				*	*	?	*	0

Opcode	Instruction	Clocks			Description
		386	286	86	
0C *ib*	OR AL,*imm8*	2	3	4	OR immediate byte to AL
0D *iw*	OR AX,*imm16*	2	3	4	OR immediate word to AX

0D *id*	OR EAX,*imm32*	2			OR immediate dword to EAX
80 */1 ib*	OR *r/m8,imm8*	2/7	3/7	4/17+EA	OR immediate byte to *r/mbyte*
81 */1 iw*	OR *r/m16,imm16*	2/7	3/7	4/17+EA	OR immediate word to *r/m* word
81 */1 id*	OR *r/m32,imm32*	2/7			OR immediate dword to *r/m* dword
83 */1 ib*	OR *r/m16,imm8*	2/7			OR sign-extended immediate byte with *r/m* word
83 */1 ib*	OR *r/m32,imm8*	2/7			OR sign-extended immediate byte with *r/m* dword
08 */r*	OR *r/m8,r8*	2/6	2/7	3/16+EA	OR byte register to *r/m* byte
09 */r*	OR *r/m16,r16*	2/6	2/7	3/16+EA	OR word register to *r/m* word
09 */r*	OR *r/m32,r32*	2/6			OR dword register to *r/m* dword
0A */r*	OR *r8,r/m8*	2/7	2/7	3/9+EA	OR byte register to *r/m* byte
0B */r*	OR *r16,r/m16*	2/7	2/7	3/9+EA	OR word register to *r/m* word
0B */r*	OR *r32,r/m32*	2/7			OR dword register to *r/m* word

OR computes the inclusive OR of its two operands and places the result in the first operand. Each bit of the result is 0 if both corresponding bits of the operands are 0; otherwise, each bit is 1.

OUT

Output to Port

O D I T S Z A P C

Opcode	Instruction	Clocks			Description
		386	286	86	
E6 *ib*	OUT *imm8*,AL	10,pm=4*/24**	3	10	Output byte AL to immediate port number
E7 *ib*	OUT *imm8*,AX	10,pm=4*/24**	3	10	Output word AX to immediate port number
E7 *ib*	OUT *imm8*,EAX	10,pm=4*/25**			Output dword EAX to immediate port number
EE	OUT DX,AL	11,pm=5*/25**	3	8	Output byte AL to port number in DX
EF	OUT DX,AX	11,pm=5*/25**	3	8	Output word AX to port number in DX
EF	OUT DX,EAX	11,pm=5*/25**			Output dword EAX to port number in DX

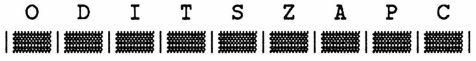
* If CPL ≤ IOPL
** If CPL > IOPL or if in virtual 8086 mode

OUT transfers a data byte or data word from the register (AL, AX, or EAX) given as the second operand to the output port numbered by the first operand. Output to any port from 0 to 65535 is performed by placing the port number in the DX register and then using an OUT instruction with DX as the first operand. If the instruction contains an eight-bit port ID, that value is zero-extended to 16 bits.

OUTS
OUTSB
OUTSW
OUTSD

Output String to Port

OUTS/OUTSB/OUTSW 80186/286/386 only
OUTSD 80386 only



Opcode	Instruction	Clocks			Description
		386	286	86	
6E	OUTS DX,*r/m8*	14,*pm*= 8*/28**	5		Output byte [(E)SI] to port in DX
6F	OUTS DX,*r/m16*	14,*pm*= 8*/28**	5		Output word [(E)SI] to port in DX
6F	OUTS DX,*r/m32*, EAX	14,*pm*= 8*/28**			Output dword [(E)SI] to port in DX
6E	OUTSB	14,*pm*= 8*/28**	5		Output byte DS:[(E)SI] to port in DX
6F	OUTSW	14,*pm*= 8*/28**	5		Output word DS:[(E)SI] to port number in DX
6F	OUTSD	14,*pm*= 8*/28**			Output dword DS:[(E)SI] to port in DX

OUTS transfers data from the memory byte, word, or doubleword at the source-index register to the output port addressed by the DX register. If the address-size attribute for this instruction is 16 bits, SI is used for the source-index register; otherwise, the address-size attribute is 32 bits, and ESI is used for the source-index register.

OUTS does not allow specification of the port number as an immediate value. The port must be addressed through the DX register value. Load the correct value into DX before executing the OUTS instruction.

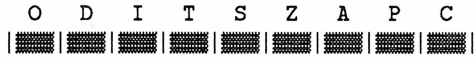
The address of the source data is determined by the contents of source-index register. Load the correct index value into SI or ESI before executing the OUTS instruction.

After the transfer, source-index register is advanced automatically. If the direction flag is 0 (CLD was executed), the source-index register is incremented; if the direction flag is 1 (STD was executed), it is decremented. The amount of the increment or decrement is 1 if a byte is output, 2 if a word is output, or 4 if a doubleword is output.

OUTSB, OUTSW, and OUTSD are synonyms for the byte, word, and doubleword OUTS instructions. OUTS can be preceded by the REP prefix for block output of CX bytes or words. Refer to the REP instruction for details on this operation.

POP

Pop a Word from the Stack



Opcode	Instruction	Clocks			Description
		386	286	86	
8F /0	POP *m16*	5	5	17+EA	Pop top of stack into memory word
8F /0	POP *m32*	5			Pop top of stack into memory dword
58+*rw*	POP *r16*	4	5	8	Pop top of stack into word register
58+*rd*	POP *r32*	4			Pop top of stack into dword register
1F	POP DS	7,*pm*=21	5,*pm*=20	8	Pop top of stack into DS
07	POP ES	7,*pm*=21	5,*pm*=20	8	Pop top of stack into ES
17	POP SS	7,*pm*=21	5,*pm*=20	8	Pop top of stack into SS
0F A1	POP FS	7,*pm*=21			Pop top of stack into FS
0F A9	POP GS	7,*pm*=21			Pop top of stack into GS

POP replaces the previous contents of the memory, the register, or the segment register operand with the word on the top of the stack, addressed by SS:SP (address-size attribute of 16 bits) or SS:ESP (address-size attribute of 32 bits). The stack pointer SP is incremented by 2 for an

operand-size of 16 bits or by 4 for an operand-size of 32 bits. It then points to the new top of stack.

POP CS is not an instruction. Popping from the stack into the CS register is accomplished with a RET instruction.

If the destination operand is a segment register (DS, ES, FS, GS, or SS), the value popped must be a selector. In protected mode, loading the selector initiates automatic loading of the descriptor information associated with that selector into the hidden part of the segment register; loading also initiates validation of both the selector and the descriptor information.

A null value (0000-0003) may be popped into the DS, ES, FS, or GS register without causing a protection exception. An attempt to reference a segment whose corresponding segment register is loaded with a null value causes a #GP(0) exception. No memory reference occurs. The saved value of the segment register is null.

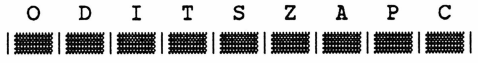
A POP SS instruction inhibits all interrupts, including NMI, until after execution of the next instruction. This allows sequential execution of POP SS and POP eSP instructions without danger of having an invalid stack during an interrupt. However, use of the LSS instruction is the preferred method of loading the SS and eSP registers.

Note: Turbo Assembler extends the syntax of the POP instruction to facilitate popping many registers in sequence. This feature does not actually affect the code generated.

POPA
POPAD

Pop All General Registers

POPA 80186/286/386 only
POPAD 80386 only



Opcode	Instruction	Clocks			Description
		386	286	86	
61	POPA	24	19		Pop DI, SI, BP, SP, BX, DX, CX, and AX
61	POPAD	24			Pop EDI, ESI, EBP, ESP,EBX, EDX, ECX, and EAX

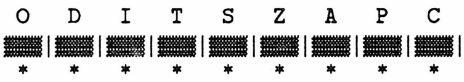
POPA pops the eight 16-bit general registers. However, the SP value is discarded instead of loaded into SP. POPA reverses a previous PUSHA, restoring the general registers to their values before PUSHA was executed. The first register popped is DI.

POPAD pops the eight 32-bit general registers. The ESP value is discarded instead of loaded into ESP. POPAD reverses the previous PUSHAD, restoring the general registers to their values before PUSHAD was executed. The first register popped is EDI.

POPF
POPFD

Pop from Stack into FLAGS or EFLAGS Register

POPFD 80386 only



Opcode	Instruction	Clocks			Description
		386	286	86	
9D	POPF	5	5	8	Pop top of stack FLAGS
9D	POPFD	5			Pop top of stack into EFLAGS

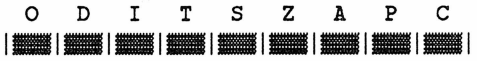
POPF/POPFD pops the word or doubleword on the top of the stack and stores the value in the flags register. If the operand-size attribute of the instruction is 16 bits, then a word is popped and the value is stored in FLAGS. If the operand-size attribute is 32 bits, then a doubleword is popped and the value is stored in EFLAGS.

Note that bits 16 and 17 of EFLAGS, called VM and RF, respectively, are not affected by POPF or POPFD.

The I/O privilege level is altered only when executing at privilege level 0. The interrupt flag is altered only when executing at a level at least as privileged as the I/O privilege level. (Real-address mode is equivalent to privilege level 0.) If a POPF instruction is executed with insufficient privilege, an exception does not occur, but the privileged bits do not change.

PUSH

Push Operand onto the Stack



Opcode	Instruction	Clocks			Description
		386	286	86	
FF /6	PUSH *m16*	5	5	16+EA	Push memory word
FF /6	PUSH *m32*	5			Push memory dword
50+ /r	PUSH *r16*	2	3	11	Push register word
50+ /r	PUSH *r32*	2			Push register dword
6A	PUSH *imm8*	2	3		Push immediate byte
68	PUSH *imm16*	2	3		Push immediate word
68	PUSH *imm32*	2			Push immediate dword
0E	PUSH CS	2	3	10	Push CS
16	PUSH SS	2	3	10	Push SS
1E	PUSH DS	2	3	10	Push DS
06	PUSH ES	2	3	10	Push ES
0F A0	PUSH FS	2			Push FS
0F A8	PUSH GS	2			Push GS

PUSH decrements the stack pointer by 2 if the operand-size attribute of the instruction is 16 bits; otherwise, it decrements the stack pointer by 4. PUSH then places the operand on the new top of stack, which is pointed to by the stack pointer.

The 80386 PUSH eSP instruction pushes the value of the eSP as it existed before the instruction. The 80286 PUSH SP instruction also pushes the value of SP as it existed before the instruction. This differs from the 8086, where PUSH SP pushes the new value (decremented by 2).

Note: Turbo Assembler extends the syntax of the PUSH instruction to facilitate pushing many registers in sequence. This feature does not actually affect the code generated.

PUSHA
PUSHAD

Push All General Registers

PUSHA 80186/286/386 only
PUSHAD 80386 only

Opcode	Instruction	Clocks			Description
		386	286	86	
60	PUSHA	18	17		Push AX,CX,DX,BX,original SP,BP,SI, and DI
60	PUSHAD	18			Push EAX,ECX,EDX,EBX, original ESP,EBP,ESI and EDI

PUSHA and PUSHAD save the 16-bit or 32-bit general registers, respectively, on the stack. PUSHA decrements the stack pointer (SP) by 16 to hold the eight word values. PUSHAD decrements the stack pointer (ESP) by 32 to hold the eight doubleword values. Because the registers are pushed onto the stack in the order in which they were given, they appear in the 16 or 32 new stack bytes in reverse order. The last register pushed is DI or EDI.

PUSHF
PUSHFD

Push Flags Register onto the Stack

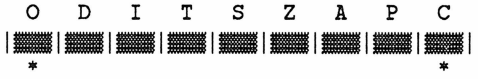
PUSHFD 80386 only

Opcode	Instruction	Clocks			Description
		386	286	86	
9C	PUSHF	4	3	10	Push FLAGS

PUSHF decrements the stack pointer by 2 and copies the FLAGS register to the new top of stack; PUSHFD decrements the stack pointer by 4, and the 80386 EFLAGS register is copied to the new top of stack which is pointed to by SS:eSP.

RCL
RCR
ROL
ROR

Rotate



Opcode	Instruction	386	286	86	Description
D0 /2	RCL *r/m8*,1	9/10	2/7	2/15+EA	Rotate 9 bits (CF,*r/m* byte)left once
D2 /2	RCL *r/m8*,CL	9/10	5/8	8+4 per bit/ (20+4 per bit)+EA	Rotate 9 bits (CF,*r/m* byte)left CL times
C0 /2 *ib*	RCL *r/m8,imm8*	9/10	5/8		Rotate 9 bits (CF,*r/m* byte) left *imm8* times
D1 /2	RCL *r/m16*,1	9/10	2/7	2/15+EA	Rotate 17 bits (CF,*r/m* word) left once
D3 /2	RCL *r/m16*,CL	9/10	5/8	8+4 per bit/ (20+4 per bit)+EA	Rotate 17 bits (CF, *r/m* word) left CL times
C1 /2 *ib*	RCL *r/m16,imm8*	9/10	5/8		Rotate 17 bits (CF,*r/m* word) left *imm8* times
D1 /2	RCL *r/m32*,1	9/10			Rotate 33 bits (CF,*r/m* dword) left once
D3 /2	RCL *r/m32*,CL	9/10			Rotate 33 bits (CF,*r/m* dword) left CL times
C1 /2 *ib*	RCL *r/m32,imm8*	9/10			Rotate 33 bits (CF,*r/m* dword) left, *imm8* times
D0 /3	RCR *r/m8*,1	9/10	2/7	2/15+EA	Rotate 9 bits (CF,*r/m* byte) right once
D2 /3	RCR *r/m8*,CL	9/10	5/8	8+4 per bit/ (20+4 per bit)+EA	Rotate 9 bits (CF,*r/m* byte) right CL times

C0 /3 *ib*	RCR *r/m8,imm8*	9/10	5/8		Rotate 9 bits (CF,*r/m* byte) right *imm8* times
D1 /3	RCR *r/m16*,1	9/10	2/7	2/15+EA	Rotate 17 bits (CF,*r/m* word) right once
D3 /3	RCR *r/m16*,CL	9/10	5/8	8+4 per bit/ (20+4 per bit)+EA	Rotate 17 bits (CF,*r/m* word) right CL times
C1 /3 *ib*	RCR *r/m16,imm8*	9/10	5/8		Rotate 17 bits (CF,*r/m* word) right *imm8* times
D1 /3	RCR *r/m32*,1	9/10			Rotate 33 bits (CF,*r/m* dword) right once
D3 /3	RCR *r/m32*,CL	9/10			Rotate 33 bits (CF,*r/m* dword) right CL times
C1 /3 *ib*	RCR *r/m32,imm8*	9/10			Rotate 33 bits (CF,*r/m* dword) right *imm8* times
D0 /0	ROL *r/m8*1	3/7	2/7	2/15+EA	Rotate 8 bits *r/m* byte left once
D2 /0	ROL *r/m8*CL	3/7	5/8	8+4 per bit/ (20+4 per bit)+EA	Rotate 8 bits *r/m* byte left CL times
C0 /0 *ib*	ROL *r/m8,imm8*	3/7	5/8		Rotate 8 bits *r/m* byte left *imm8* times
D1 /0	ROL *r/m16*,1	3/7	2/7	2/15+EA	Rotate 16 bits *r/m* word left once
D3 /0	ROL *r/m16*,CL	3/7	5/8	8+4 per bit/ (20+4 per bit)+EA	Rotate 16 bits *r/m* word left CL times
C1 /0 *ib*	ROL *r/m16,imm8*	3/7	5/8		Rotate 16 bit *r/m* word left *imm8* times
D1 /0	ROL *r/m32*,1	3/7			Rotate 32 bits *r/m* dword left once
D3 /0	ROL *r/m32*,CL	3/7			Rotate 32 bits *r/m* dword left CL times
C1 /0 *ib*	ROL *r/m32,imm8*	3/7			Rotate 32 bits *r/m* dword left *imm8* times
D0 /1	ROR *r/m8*,1	3/7	2/7	2/15+EA	Rotate 8 bits *r/m* byte right once
D2 /1	ROR *r/m8*,CL	3/7	5/8	8+4 per bit/ (20+4 per bit)+EA	Rotate 8 bits *r/m* byte right CL times
C0 /1 db	ROR *r/m8,imm8*	3/7	5/8		Rotate 8 bits *r/m* byte right *imm8* times
D1 /1	ROR *r/m16*,1	3/7	2/7	2/15+EA	Rotate 16 bit *r/m* word right once
D3 /1	ROR *r/m16*,CL	3/7	5/8	8+4 per bit/ (20+4 per bit)+EA	Rotate 16 bit *r/m* word right CL once
C1 /1 *ib*	ROR *r/m16,imm8*	3/7	5/8		Rotate 16 bit *r/m* word right *imm8* times
D1 /1	ROR *r/m16*,1	3/7			Rotate 32 bits *r/m* dword right once
D3 /1	ROR *r/m32*,CL	3/7			Rotate 32 bits *r/m* dword right CL times
C1 /1 *ib*	ROR *r/m32,imm8*	3/7			Rotate 32 bits *r/m* dword right *imm8* times

*Add 1 clock to the times shown for each rotate made (80286 only).

Each rotate instruction shifts the bits of the register or memory operand given. The left rotate instructions shift all the bits upward, except for the top bit, which is returned to the bottom. The right rotate instructions do the reverse: the bits shift downward until the bottom bit arrives at the top.

For the RCL and RCR instructions, the carry flag is part of the rotated quantity. RCL shifts the carry flag into the bottom bit and shifts the top bit into the carry flag; RCR shifts the carry flag into the top bit and shifts the bottom bit into the carry flag. For the ROL and ROR instructions, the original value of the carry flag is not a part of the result, but the carry flag receives a copy of the bit that was shifted from one end to the other.

The rotate is repeated the number of times indicated by the second operand, which is either an immediate number or the contents of the CL register. To reduce the maximum instruction execution time, the 80286/80386 does not allow rotation counts greater than 31. If a rotation count greater than 31 is attempted, only the bottom five bits of the rotation are used. The 8086 does not mask rotation counts. The 80386 in Virtual 8086 Mode does mask rotation counts.

The overflow flag is defined only for the single-rotate forms of the instructions (second operand = 1). It is undefined in all other cases. For left shifts/rotates, the CF bit after the shift is XORed with the high order result bit. For right shifts/rotates, the high-order two bits of the result are XORed to get OF.

REP
REPE
REPZ
REPNE
REPNZ

Repeat Following String Operation

O	D	I	T	S	Z	A	P	C
					*			

ZF by REP CMPS and REP SCAS as described next.

Opcode	Instruction	Clocks			Description
		386	286	86	

F3 6C	REP INS *r/m8*,DX	13+6*(E)CX, *pm*=7+6*(E)CX*1/27+6*(E)CX*2	5+4*CX		Input (E)CX words from port DX into ES:[(E)DI]
F3 6D	REP INS *r/m16*,DX	13+6*(E)CX, *pm*=7+6*(E)CX*1/27+6*(E)CX*2	5+4*CX		Input (E)CX bytes from port DX into ES:[(E)DI]
F3 6D	REP INS *r/m32*,DX	13+6*(E)CX, *pm*=7+6*(E)CX*1/27+6*(E)CX*2			Input (E)CX dwords from port DX into ES:[(E)DI]
F3 A4	REP MOVS *m8,m8*	5+4*(E)CX	5+4*CX	9+17*CX	Move (E)CX bytes from [(E)SI] to ES:[(E)DI]
F3 A5	REP MOVS *m16,m16*	5+4*(E)CX	5+4*CX	9+17*CX	Move (E)CX words from [(E)SI] to ES:[(E)DI]
F3 A5	REP MOVS *m32,m32*	5+4*(E)CX			Move (E)CX dwords from [(E)SI] to ES:[(E)DI]
F3 6E	REP OUTS DX,*r/m8*	5+12*(E)CX, *pm*=6+5*(E)CX*1/26+5*(E)CX*2	5+4*CX		Output (E)CX bytes from [(E)SI] to port DX
F3 6F	REP OUTS DX,*r/m16*	5+12*(E)CX, *pm*=6+5*(E)CX*1/26+5*(E)CX*2	5+4*CX		Output (E)CX words from [(E)SI] to port DX
F3 6F	REP OUTS DX,*r/m32*	5+12*(E)CX, *pm*=6+5*(E)CX*1/26+5*(E)CX*2			Output(E)CX dwords from [(E)SI] to port DX
F3 AA	REP STOS *m8*	5+5*(E)CX	4+3*CX	9+10*CX	Fill (E)CX bytes at ES:[(E)DI] with AL
F3 AB	REP STOS *m16*	5+5*(E)CX	4+3*CX	9+10*CX	Fill (E)CX words at ES:[(E)DI] with AX
F3 AB	REP STOS *m32*	5+5*(E)CX			Fill (E)CX dwords at ES:[(E)DI] with EAX
F3 A6	REPE CMPS *m8,m8*	5+9*N	5+9*N	9+22*N	Find nonmatching bytes in ES:[(E)DI] and [(E)SI]
F3 A7	REPE CMPS *m16,m16*	5+9*N	5+9*N	9+22*N	Find nonmatching words in ES:[(E)DI] and [(E)SI]
F3 A7	REPE CMPS *m32,m32*	5+9*N			Find nonmatching dwords in ES:[(E)DI] and [(E)SI]
F3 AE	REPE SCAS *m8*	5+8*N	5+8*N	9+15*N	Find non-AL byte starting at ES:[(E)DI]
F3 AF	EPE SCAS *m16*	5+8*N	5+8*N	9+15*N	Find non-AX word starting at ES:[(E)DI]
F3 AF	REPE SCAS *m32*	5+8*N			Find non-EAX dword starting at ES:[(E)DI]
F2 A6	REPNE CMPS *m8,m8*	5+9*N	5+9*N	9+22*N	Find matching bytes in ES:[(E)DI] and [(E)SI]
F2 A7	REPNE CMPS *m16,m16*	5+9*N	5+9*N	9+22*N	Find matching words in ES:[(E)DI] and [(E)SI]
F2 A7	REPNE CMPS *m32,m32*	5+9*N			Find matching dwords in ES:[(E)DI] and [(E)SI]
F2 AE	REPNE SCAS *m8*	5+8*N	5+8*N	9+15*N	Find AL, starting at ES:[(E)DI]

F2 AF	REPNE SCAS *m16*	5+8*N	5+8*N	9+15*N	Find AX, starting at ES:[(E)DI]
F2 AF	REPNE SCAS *m32*	5+8*N			Find EAX, starting at ES:[(E)DI]

*1 If CPL ≤ IOPL
*2 If CPL > IOPL or if in virtual 8086 mode

REP, REPE (repeat while equal), and REPNE (repeat while not equal) are prefixes that are applied to string operations. Each prefix causes the string instruction that follows to be repeated the number of times indicated in the count register or (for REPE and REPNE) until the indicated condition in the zero flag is no longer met.

Synonymous forms of REPE and REPNE are REPZ and REPNZ, respectively.

The REP prefixes apply only to one string instruction at a time. To repeat a block of instructions, use the LOOP instruction or another looping construct.

The precise action for each iteration is as follows:

1. If the address-size attribute is 16 bits, use CX for the count register; if the address-size attribute is 32 bits, use ECX for the count register.
2. Check CX. If it is zero, exit the iteration, and move to the next instruction.
3. Acknowledge any pending interrupts.
4. Perform the string operation once.
5. Decrement CX or ECX by one; no flags are modified.
6. Check the zero flag if the string operation is SCAS or CMPS. If the repeat condition does not hold, exit the iteration and move to the next instruction. Exit the iteration if the prefix is REPE and ZF is 0 (the last comparison was not equal), or if the prefix is REPNE and ZF is one (the last comparison was equal).
7. Return to step 1 for the next iteration.

Repeated CMPS and SCAS instructions can be exited if the count is exhausted or if the zero flag fails the repeat condition. These two cases can be distinguished by using either the JCXZ instruction, or by using the conditional jumps that test the zero flag (JZ, JNZ, and JNE).

RET

Return from Procedure

O	D	I	T	S	Z	A	P	C

Opcode	Instruction	Clocks			Description
		386	286	86	
C3	RET	10+*m*	11	16	Return (near) to caller
CB	RET	18+*m*,*pm*=32+*m*	15,*pm*=25	26	Return (far) to caller, same privilege
CB	RET	*pm*=68	55		Return (far), lesser privilege, switch stacks
C2 *iw*	RET *imm16*	10+*m*	11	20	Return (near), pop *imm16* bytes of parameters
CA *iw*	RET *imm16*	18+*m*,*pm*=32+*m*	15,*pm*=25	25	Return (far), same privilege, pop *imm16* bytes
CA *iw*	RET *imm16*	*pm*=68	55		Return (far), lesser privilege, pop *imm16* bytes

RET transfers control to a return address located on the stack. The address is usually placed on the stack by a CALL instruction, and the return is made to the instruction that follows the CALL.

The optional numeric parameter to RET gives the number of stack bytes (OperandMode = 16) or words (OperandMode = 32) to be released after the return address is popped. These items are typically used as input parameters to the procedure called.

For the intrasegment (near) return, the address on the stack is a segment offset, which is popped into the instruction pointer. The CS register is unchanged. For the intersegment (far) return, the address on the stack is a long pointer. The offset is popped first, followed by the selector.

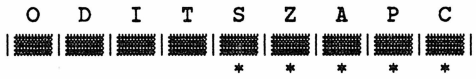
In real mode, CS and IP are loaded directly. In Protected Mode, an intersegment return causes the processor to check the descriptor addressed by the return selector. The AR byte of the descriptor must indicate a code segment of equal or lesser privilege (or greater or equal numeric value) than the current privilege level. Returns to a lesser privilege level cause the stack to be reloaded from the value saved beyond the parameter block.

The DS, ES, FS, and GS segment registers can be set to 0 by the RET instruction during an interlevel transfer. If these registers refer to

segments that cannot be used by the new privilege level, they are set to 0 to prevent unauthorized access from the new privilege level.

SAHF

Store AH into Flags

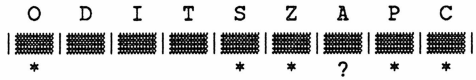


Opcode	Instruction	Clocks			Description
		386	286	86	
9E	SAHF	3	2	4	Store AH flags SF ZF xx AF xx PF xx CF

SAHF loads the flags listed above with values from the AH register, from bits 7, 6, 4, 2 and 0, respectively.

SAL
SAR
SHL
SHR

Shift Instructions



Opcode	Instruction	Clocks			Description
		386	286*	86	
D0 /4	SAL *r/m8*,1	3/7	2/7	2/15+EA	Multiply *r/m* byte by 2, once
D2 /4	SAL *r/m8*,CL	3/7	5/8	8+4 per bit/ (20+4 per bit)+EA	Multiply *r/m* byte by 2, CL times
C0 /4 *ib*	SAL *r/m8,imm8*	3/7	5/8		Multiply *r/m* byte by 2, *imm8* times
D1 /4	SAL *r/m16*,1	3/7	2/7	2/15+EA	Multiply *r/m* word by 2, once
D3 /4	SAL *r/m16*,CL	3/7	5/8	8+4 per bit	Multiply *r/m* word by 2,

				(20+4 per bit)+EA	CL times
C1 /4 *ib*	SAL *r/m*16,*imm8*	3/7	5/8		Multiply *r/m* word by 2, *imm8* times
D1 /4	SAL *r/m*32,1	3/7			Multiply *r/m* dword by 2, once
D3 /4	SAL *r/m*32,CL	3/7			Multiply *r/m* dword by 2, CL times
C1 /4 *ib*	SAL *r/m*32,*imm8*	3/7			Multiply *r/m* dword by 2, *imm8* times
D0 /7	SAR *r/m*8,1	3/7	2/7	2/15+EA	Signed divide** *r/m* byte by 2, once
D2 /7	SAR *r/m*8,CL	3/7	5/8	8+4 per bit (20+4 per bit)+EA	Signed divide** *r/m* byte by 2, CL times
C0 /7 *ib*	SAR *r/m*8,*imm8*	3/7	5/8		Signed divide** *r/m* byte by 2, *imm8* times
D1 /7	SAR *r/m*16,1	3/7	2/7	2/15+EA	Signed divide** *r/m* word by 2, once
D3 /7	SAR *r/m*16,CL	3/7	5/8	8+4 per bit (20+4 per bit)+EA	Signed divide** *r/m* word by 2, CL times
C1 /7 *ib*	SAR *r/m*16,*imm8*	3/7	5/8		Signed divide** *r/m* word by 2, *imm8* times
D1 /7	SAR *r/m*32,1	3/7			Signed divide** *r/m* dword by 2, once
D3 /7	SAR *r/m*32,CL	3/7			Signed divide** *r/m* dword by 2,CL times
C1 /7	SAR *r/m*32,*imm8*	3/7			Signed divide** *r/m* dword by 2, *imm8* times
D0 /4	SHL *r/m*8,1	3/7	2/7	2/15+EA	Multiply *r/m* byte by 2, once
D2 /4	SHL *r/m*8,CL	3/7	5/8	8+4 per bit (20+4 per bit)+EA	Multiply *r/m* byte by 2, CL times
C0 /4 *ib*	SHL *r/m*8,*imm8*	3/7	5/8		Multiply *r/m* byte by 2, *imm8* times
D1 /4	SHL *r/m*16,1	3/7	2/7	2/15+EA	Multiply *r/m* word by 2, once
D3 /4	SHL *r/m*16,CL	3/7	5/8	8+4 per bit (20+4 per bit)+EA	Multiply *r/m* word by 2, CL times
C1 /4 *ib*	SHL *r/m*16,*imm8*	3/7	5/8		Multiply *r/m* word by 2, *imm8* times
D1 /4	SHL *r/m*32,1	3/7			Multiply *r/m* dword by 2, once
D3 /4	SHL *r/m*32,CL	3/7			Multiply *r/m* dword by 2, CL times
C1 /4	SHL *r/m*32,*imm8*	3/7			Multiply *r/m* dword by 2, *imm8*times
D0 /5	SHR *r/m*8,1	3/7	2/7	2/15+EA	Unsigned divide *r/m* byte by 2, once
D2 /5	SHR *r/m*8,CL	3/7	5/8	8+4 per bit (20+4 per bit)+EA	Unsigned divide *r/m* byte by 2, CL times
C0 /5 *ib*	SHR *r/m*8,*imm8*	3/7	5/8		Unsigned divide *r/m* byte by 2, *imm8* times

D1 /5	SHR *r/m*16,1	3/7	2/7	2/15+EA	Unsigned divide *r/m* word by 2, once
D3 /5	SHR *r/m*16,CL	3/7	5/8	8+4 per bit (20+4 per bit)+EA	Unsigned divide *r/m* word by 2, CL times
C1 /5 *ib*	SHR *r/m*16,*imm8*	3/7	5/8		Unsigned divide *r/m* word by 2, *imm8* times
D1 /5	SHR *r/m*32,1	3/7			Unsigned divide *r/m* dword by 2, once
D3 /5	SHR *r/m*32,CL	3/7			Unsigned divide *r/m* dword by 2, CL times
C1 /5 *ib*	SHR *r/m*32,*imm8*	3/7			Unsigned divide *r/m* dword by 2, *imm8* times

*Not the same division as IDIV; rounding is toward negative infinity.
**Add 1 clock to the times shown for each shift performed.

SAL (or its synonym, SHL) shifts the bits of the operand upward. The high-order bit is shifted into the carry flag, and the low-order bit is set to 0.

SAR and SHR shift the bits of the operand downward. The low-order bit is shifted into the carry flag. The effect is to divide the operand by 2. SAR performs a signed divide with rounding toward negative infinity (not the same as IDIV); the high-order bit remains the same. SHR performs an unsigned divide; the high-order bit is set to 0.

The shift is repeated the number of times indicated by the second operand, which is either an immediate number or the contents of the CL register. To reduce the maximum execution time, the 80286/80386 does not allow shift counts greater than 31. If a shift count greater than 31 is attempted, only the bottom five bits of the shift count are used. (The 8086 uses all eight bits of the shift count.)

The overflow flag is set only if the single-shift forms of the instructions are used. For left shifts, OF is set to 0 if the high bit of the answer is the same as the result of the carry flag (i.e., the top two bits of the original operand were the same); OF is set to 1 if they are different. For SAR, OF is set to 0 for all single shifts. For SHR, OF is set to the high-order bit of the original operand.

SBB

Integer Subtraction with Borrow

		*		*	*	*	*	*
Opcode	**Instruction**		**Clocks**			**Description**		
		386	286	86				
1C *ib*	SBB AL,*imm8*	2	3	4		Subtract with borrow immediate byte from AL		
1D *iw*	SBB AX, *imm16*	2	3	4		Subtract with borrow immediate word from AX		
1D *id*	SBB EAX, *imm32*	2				Subtract with borrow immediate dword from EAX		
80 /3 *ib*	SBB *r/m8,imm8*	2/7	3/7	4/17+EA		Subtract with borrow immediate byte from *r/m* byte		
81 /3 *iw*	SBB *r/m16,imm16*	2/7	3/7	4/17+EA		Subtract with borrow immediate word from *r/m* word		
81 /3 *id*	SBB *r/m32,imm32*	2/7				Subtract with borrow immediate dword from *r/m* dword		
83 /3 *ib*	SBB *r/m16,imm8*	2/7	3/7	4/17+EA		Subtract with borrow sign-extended immediate byte from *r/m* word		
83 /3 *ib*	SBB *r/m32,imm8*	2/7				Subtract with borrow sign-extended immediate byte from *r/m* dword		
18 /*r*	SBB *r/m8,r8*	2/6	2/7	3/16+EA		Subtract with borrow byte register from *r/m* byte		
19 /*r*	SBB *r/m16,r16*	2/6	2/7	3/16+EA		Subtract with borrow word register from *r/m* word		
19 /*r*	SBB *r/m32,r32*	2/6				Subtract with borrow dword register from *r/m* dword		
1A /*r*	SBB *r8,r/m8*	2/7	2/7	3/9+EA		Subtract with borrow byte register from *r/m* byte		
1B /*r*	SBB *r16,r/m16*	2/7	2/7	3/9+EA		Subtract with borrow word register from *r/m* word		
1B /*r*	SBB *r32,r/m32*	2/7				Subtract with borrow dword register from *r/m* dword		

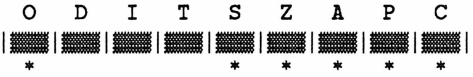
SBB adds the second operand (DEST) to the carry flag (CF) and subtracts the result from the first operand (SRC). The result of the subtraction is assigned to the first operand (DEST), and the flags are set accordingly.

When an immediate byte value is subtracted from a word operand, the immediate value is first sign-extended.

SCAS
SCASB
SCASW
SCASD

Compare String Data

SCASD 80386 only



Opcode	Instruction	Clocks			Description
		386	286	86	
AE	SCAS *m8*	7	7	15	Compare bytes AL – ES:[DI], update (E)DI
AF	SCAS *m16*	7	7	15	Compare words AX – ES:[DI], update (E)DI
AF	SCAS *m32*	7			Compare dwords EAX – ES:[DI], update (E)DI
AE	SCASB	7	7	15	Compare bytes AL – ES:[DI], update (E)DI
AF	SCASW	7	7	15	Compare words AX – ES:[DI], update (E)DI
AF	SCASD	7			Compare dwords EAX – ES:[DI], update (E)DI

SCAS subtracts the memory byte or word at the destination register from the AL, AX or EAX register. The result is discarded; only the flags are set. The operand must be addressable from the ES segment; no segment override is possible.

If the address-size attribute for this instruction is 16 bits, DI is used as the destination register; otherwise, the address-size attribute is 32 bits and EDI is used.

The address of the memory data being compared is determined solely by the contents of the destination register, not by the operand to SCAS. The operand validates ES segment addressability and determines the data type. Load the correct index value into DI or EDI before executing SCAS.

After the comparison is made, the destination register is automatically updated. If the direction flag is 0 (CLD was executed), the destination register is incremented; if the direction flag is 1 (STD was executed), it is

decremented. The increments or decrements are by 1 if bytes are compared, by 2 if words are compared, or by 4 if doublewords are compared.

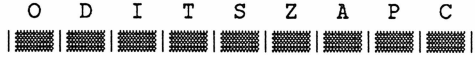
SCASB, SCASW, and SCASD are synonyms for the byte, word and doubleword SCAS instructions that don't require operands. They are simpler to code, but provide no type or segment checking.

SCAS can be preceded by the REPE or REPNE prefix for a block search of CX or ECX bytes or words. Refer to the REP instruction for further details.

SETcc

Byte Set on Condition

80386 only



Opcode	Instruction	Clocks			Description
		386	286	86	
0F 97	SETA *r/m8*	4/5			Set byte if above (CF=0 and ZF =0)
0F 93	SETAE *r/m8*	4/5			Set byte if above or equal (CF=0)
0F 92	SETB *r/m8*	4/5			Set byte if below (CF=1)
0F 96	SETBE *r/m8*	4/5			Set byte if below or equal (CF=1 or ZF=1)
0F 92	SETC *r/m8*	4/5			Set if carry (CF=1)
0F 94	SETE *r/m8*	4/5			Set byte if equal (ZF=1)
0F 9F	SETG *r/m8*	4/5			Set byte if greater (ZF=0 or SF=OF)
0F 9D	SETGE *r/m8*	4/5			Set byte if greater or equal (SF=OF)
0F 9C	SETL *r/m8*	4/5			Set byte if less (SF<>OF)
0F 9E	SETLE *r/m8*	4/5			Set byte if less or equal (ZF=1 and SF<>OF)
0F 96	SETNA *r/m8*	4/5			Set byte if not above (CF=1)
0F 92	SETNAE *r/m8*	4/5			Set byte if not above or equal (CF=1)
0F 93	SETNB *r/m8*	4/5			Set byte if not below (CF=0)
0F 97	SETNBE *r/m8*	4/5			Set byte if not below or equal (CF=0 and ZF=0)
0F 93	SETNC *r/m8*	4/5			Set byte if not carry (CF=0)
0F 95	SETNE *r/m8*	4/5			Set byte if not equal (ZF=0)
0F 9E	SETNG *r/m8*	4/5			Set byte if not greater (ZF=1 or SF<>OF)

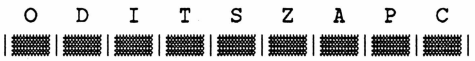
0F 9C	SETNGE *r/m8*	4/5	Set byte if not greater or equal (SF<>OF)
0F 9D	SETNL *r/m8*	4/5	Set byte if not less (SF=OF)
0F 9F	SETNLE *r/m8*	4/5	Set byte if not less or equal (ZF=1 and SF<>OF)
0F 91	SETNO *r/m8*	4/5	Set byte if not overflow (OF=0)
0F 9B	SETNP *r/m8*	4/5	Set byte if not parity (PF=0)
0F 99	SETNS *r/m8*	4/5	Set byte if not sign (SF=0)
0F 95	SETNZ *r/m8*	4/5	Set byte if not zero (ZF=0)
0F 90	SETO *r/m8*	4/5	Set byte if overflow (OF=1)
0F 9A	SETP *r/m8*	4/5	Set byte if parity (PF=1)
0F 9A	SETPE *r/m8*	4/5	Set byte if parity even (PF=1)
0F 9B	SETPO *r/m8*	4/5	Set byte if parity odd (PF=0)
0F 98	SETS *r/m8*	4/5	Set byte if sign (SF=1)
0F 94	SETZ *r/m8*	4/5	Set byte if zero (ZF=1)

SETcc stores a byte containing 1 at the destination specified by the effective address or register if the condition is met, or a 0 byte if the condition is not met.

SGDT
SIDT

Store Global/Interrupt Descriptor Table

80286/386 protected mode only



Opcode	Instruction	Clocks			Description
		386	286	86	
0F 01 /0	SGDT *m*	9	11		Store GDTR to m
0F 01 /1	SIDT *m*	9	12		Store IDTR to m

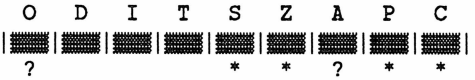
SGDT/SIDT copies the contents of the descriptor table register to the six bytes of memory indicated by the operand. The LIMIT field of the register is assigned to the first word at the effective address. If the operand-size attribute is 32 bits, the next three bytes are assigned the BASE field of the register, and the fourth byte is written with zero. The last byte is undefined. Otherwise, if the operand-size attribute is 16 bits, the next four bytes are assigned the 32-bit BASE field of the register.

SGDT and SIDT are used only in operating system software; they are not used in application programs.

SHLD

Double Precision Shift Left

80386 only



Opcode	Instruction	Clocks			Description
		386	286	86	
0F A4	SHLD *r/m16,r16,imm8*	3/7			*r/m16* gets SHL of *r/m16* concatenated with *r16*
0F A4	SHLD *r/m32,r32,imm8*	3/7			*r/m32* gets SHL of *r/m32* concatenated with *r32*
0F A5	SHLD *r/m16,r16*,CL	3/7			*r/m16* gets SHL of *r/m16* concatenated with *r16*
0F A5	SHLD *r/m32,r32*,CL	3/7			*r/m32* gets SHL of *r/m32* concatenated with *r32*

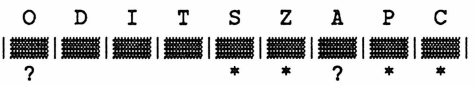
SHLD shifts the first operand provided by the *r/m* field to the left as many bits as specified by the count operand. The second operand (*r16* or *r32*) provides the bits to shift in from the right (starting with bit 0). The result is stored back into the *r/m* operand. The register remains unaltered.

The count operand is provided by either an immediate byte or the contents of the CL register. These operands are taken MODULO 32 to provide a number between 0 and 31 by which to shift. Because the bits to shift are provided by the specified registers, the operation is useful for multiprecision shifts (64 bits or more). The SF, ZF and PF flags are set according to the value of the result. CF is set to the value of the last bit shifted out. OF and AF are left undefined.

SHRD

Double Precision Shift Right

80386 only



Opcode	Instruction	Clocks			Description
		386	286	86	
OF AC	SHRD *r/m16,r16,imm8*	3/7			*r/m16* gets SHR of *r/m16* concatenated with *r16*
OF AC	SHRD *r/m32,r32,imm8*	3/7			*r/m32* gets SHR of *r/m32* concatenated with *r32*
OF AD	SHRD *r/m16,r16*,CL	3/7			*r/m16* gets SHR of *r/m16* concatenated with *r16*
OF AD	SHRD *r/m32,r32*,CL	3/7			*r/m32* gets SHR of *r/m32* concatenated with *r32*

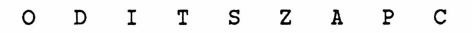
SHRD shifts the first operand provided by the *r/m* field to the right as many bits as specified by the count operand. The second operand (*r16* or *r32*) provides the bits to shift in from the left (starting with bit 31). The result is stored back into the *r/m* operand. The register remains unaltered.

The count operand is provided by either an immediate byte or the contents of the CL register. These operands are taken MODULO 32 to provide a number between 0 and 31 by which to shift. Because the bits to shift are provided by the specified register, the operation is useful for multi-precision shifts (64 bits or more). The SF, ZF and PF flags are set according to the value of the result. CF is set to the value of the last bit shifted out. OF and AF are left undefined.

SLDT

Store Local Descriptor Table Register

80286/386 protected mode only



Opcode	Instruction		Clocks		Description
		386	286	86	
0F 00 /0	SLDT *r/m16*	*pm*=2/2	2/3		Store LDTR to EA word

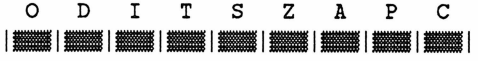
SLDT stores the Local Descriptor Table Register (LDTR) in the two-byte register or memory location indicated by the effective address operand. This register is a selector that points into the Global Descriptor Table.

SLDT is used only in operating system software. It is not used in application programs.

SMSW

Store Machine Status Word

80286/386 protected mode only

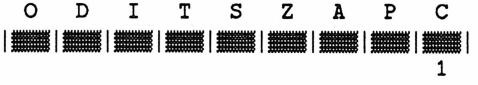


Opcode	Instruction		Clocks		Description
		386	286	86	
0F 01 /4	SMSW *r/m16*	2/3,pm=2/2	2/3		Store machine status word to EA word

SMSW stores the machine status word (part of CR0) in the two-byte register or memory location indicated by the effective address operand.

STC

Set Carry Flag



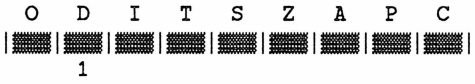
Opcode	Instruction		Clocks	Description
		386	286	86

F9	STC	2	2	2	Set carry flag

STC sets the carry flag to 1.

STD

Set Direction Flag

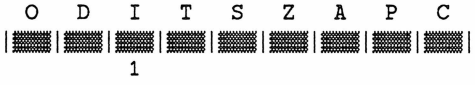


Opcode	Instruction	Clocks			Description
		386	286	86	
FD	STD	2	2	2	Set direction flag so (E)SI and/or (E)DI decrement

STD sets the direction flag to 1, causing all subsequent string operations to decrement the index registers, (E)SI and/or (E)DI, on which they operate.

STI

Set Interrupt Enable Flag



Opcode	Instruction	Clocks			Description
		386	286	86	
FB	STI	3	2	2	Set interrupt flag, interrupts enabled at the end of the next instruction

STI sets the interrupt flag to 1. The CPU then responds to external interrupts after executing the next instruction if the next instruction allows the interrupt flag to remain enabled. If external interrupts are disabled and you code STI, RET (such as at the end of a subroutine), the

RET is allowed to execute before external interrupts are recognized. Also, if external interrupts are disabled and you code STI, CLI, then external interrupts are not recognized because the CLI instruction clears the interrupt flag during its execution.

STOS
STOSB
STOSW
STOSD

Store String Data

STOSD 80386 only

O	D	I	T	S	Z	A	P	C

Opcode	Instruction	Clocks			Description
		386	286	86	
AA	STOS *m8*	4	3	11	Store AL in byte ES:[(E)DI], update (E)DI
AB	STOS *m16*	4	3	11	Store AX in word ES:[(E)DI], update (E)DI
AB	STOS *m32*	4			Store EAX in dword ES:[(E)DI], update (E)DI
AA	STOSB	4	3	11	Store AL in byte ES:[(E)DI], update (E)DI
AB	STOSW	4	3	11	Store AX in word ES:[DI], update (E)DI
AB	STOSD	4			Store EAX in dword ES:[(E)DI], update (E)DI

STOS transfers the contents of the AL, AX, or EAX register to the memory byte or word given by the destination register relative to the ES segment. The destination register is DI for an address-size attribute of 16 bits or EDI for an address-size attribute of 32 bits.

The destination operand must be addressable from the ES register. A segment override is not possible.

The address of the destination is determined by the contents of the destination register, not by the explicit operand of STOS. This operand is used only to validate ES segment addressability and to determine the data type. Load the correct index value into the destination register before executing STOS.

After the transfer is made, DI is automatically updated. If the direction flag is 0 (CLD was executed), DI is incremented; if the direction flag is 1 (STD was executed), DI is decremented. DI is incremented or decremented by 1 if a byte is stored, by 2 if a word is stored, or by 4 if a doubleword is stored.

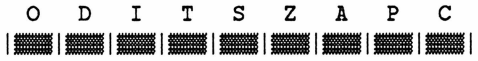
STOSB, STOSW, and STOSD are synonyms for the byte, word, and doubleword STOS instructions, that do not require an operand. They are simpler to use, but provide no type or segment checking.

STOS can be preceded by the REP prefix for a block fill of CX or ECX bytes, words, or doublewords. Refer to the REP instruction for further details.

STR

Store Task Register

80286/386 protected mode only



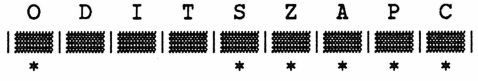
Opcode	Instruction	Clocks			Description
		386	286	86	
0F 00 /1	STR *r/m16*	pm=23/27	2/3		Load EA word into task register

The contents of the task register are copied to the two-byte register or memory location indicated by the effective address operand.

STR is used only in operating system software. It is not used in application programs.

SUB

Integer Subtraction



Opcode	Instruction	Clocks	Description

		386	286	86	
2C *ib*	SUB AL,*imm8*	2	3	4	Subtract immediate byte from AL
2D *iw*	SUB AX,*imm16*	2	3	4	Subtract immediate word from AX
2D *id*	SUB EAX,*imm32*	2			Subtract immediate dword from EAX
80 */5 ib*	SUB *r/m8,imm8*	2/7	3/7	4/17+EA	Subtract immediate byte from *r/m* byte
81 */5 iw*	SUB *r/m*16,*imm16*	2/7	3/7	4/17+EA	Subtract immediate word from *r/m* word
81 */5 id*	SUB *r/m*32,*imm32*	2/7			Subtract immediate dword from *r/m* dword
83 */5 ib*	SUB *r/m*16,*imm8*	2/7	3/7	4/17+EA	Subtract sign-extended immediate byte from *r/m* word
83 */5 ib*	SUB *r/m*32,*imm8*	2/7			Subtract sign-extended immediate byte from *r/m* dword
28 */r*	SUB *r/m8,r8*	2/6	2/7	3/16+EA	Subtract byte register from *r/m* byte
29 */r*	SUB *r/m*16,*r16*	2/6	2/7	3/16+EA	Subtract word register from *r/m* word
29 */r*	SUB *r/m*32,*r32*	2/6			Subtract dword register from *r/m* dword
2A */r*	SUB *r8,r/m*8	2/7	2/7	3/9+EA	Subtract EA byte from byte register
2B */r*	SUB *r16,r/m32*	2/7	2/7	3/9+EA	Subtract EA word from word register
2B */r*	SUB *r32,r/m32*	2/7			Subtract EA dword from dword register

SUB subtracts the second operand (SRC) from the first operand (DEST). The first operand is assigned the result of the subtraction, and the flags are set accordingly.

When an immediate byte value is subtracted from a word operand, the immediate value is first sign-extended to the size of the destination operand.

TEST

Logical Compare

O	D	I	T	S	Z	A	P	C

O		*	*	?	*	0

Opcode	Instruction	Clocks			Description
		386	286	86	
A8 *ib*	TEST AL,*imm8*	2	3	4	And immediate byte with AL
A9 *iw*	TEST AX,*imm16*	2	3	4	And immediate word with AX
A9 *id*	TEST EAX,*imm32*	2			And immediate dword with EAX
F6 /0 *ib*	TEST *r/m8,imm8*	2/5	3/6	5/11+EA	And immediate byte with *r/m* byte
F7 /0 *iw*	TEST *r/m16,imm16*	2/5	3/6	5/11+EA	And immediate word with *r/m* word
F7 /0 *id*	TEST *r/m32,imm32*	2/5			And immediate dword with *r/m* dword
84 */r*	TEST *r/m8,r8*	2/5	2/6	3/9+EA	And byte register with *r/m* byte
85 */r*	TEST *r/m16,r16*	2/5	2/6	3/9+EA	And word register with *r/m* word
85 */r*	TEST *r/m32,r32*	2/5			And dword register with *r/m* dword

TEST computes the bit-wise logical AND of its two operands. Each bit of the result is 1 if both of the corresponding bits of the operands are 1; otherwise, each bit is 0. The result of the operation is discarded and only the flags are modified.

VERR
VERW

Verify a Segment for Reading or Writing

80286/386 protected mode only

O	D	I	T	S	Z	A	P	C
					*			

Opcode	Instruction	Clocks			Description
		386	286	86	
0F 00 /4	VERR *r/m16*	pm=10/11	14/16		Set ZF=1 if segment can be read, selector in *r/m16*
0F 00 /5	VERW *r/m16*	pm=15/16	14/16		Set ZF=1 if segment can be written, selector *r/m16*

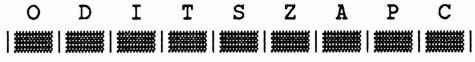
The two-byte register or memory operand of VERR and VERW contains the value of a selector. VERR and VERW determine whether the segment denoted by the selector is reachable from the current privilege level and whether the segment is readable (VERR) or writable (VERW). If the segment is accessible, the zero flag is set to 1; if the segment is not accessible, the zero flag is set to 0. To set ZF, the following conditions must be met:

- The selector must denote a descriptor within the bounds of the table (GDT or LDT); the selector must be "defined."
- The selector must denote the descriptor of a code or data segment (not that of a task state segment, LDT, or a gate).
- For VERR, the segment must be readable. For VERW, the segment must be a writable data segment.
- If the code segment is readable and conforming, the descriptor privilege level (DPL) can be any value for VERR. Otherwise, the DPL must be greater than or equal to (have less or the same privilege as) both the current privilege level and the selector's RPL.

The validation performed is the same as if the segment were loaded into DS, ES, FS, or GS, and the indicated access (read or write) were performed. The zero flag receives the result of the validation. The selector's value cannot result in a protection exception, enabling the software to anticipate possible segment access problems.

WAIT

Wait until BUSY# Pin is Inactive (HIGH)



Opcode	Instruction	Clocks			Description
		386	286	86	
9B	WAIT	6	3	4+5*n*	Wait until BUSY pin is inactive (HIGH)

WAIT suspends execution of CPU instructions until the BUSY# pin is inactive (high). The BUSY# pin is driven by the 80x87 numeric processor extension.

XCHG

Exchange Memory/Register with Register

O D I T S Z A P C

Opcode	Instruction	Clocks			Description
		386	286	86	
86 /r	XCHG *r/m8,r8*	3/5	3/5	4/17+EA	Exchange byte register with EA byte
86 /r	XCHG *r8,r/m8*	3/5	3/5	4/17+EA	Exchange byte with EA byte register
87 /r	XCHG *r/m16,r16*	3/5	3/5	4/17+EA	Exchange word register with EA word
87 /r	XCHG *r16,r/m16*	3/5	3/5	4/17+EA	Exchange word register with EA word
87 /r	XCHG *r/m32,r32*	3/5			Exchange dword register with EA dword
87 /r	XCHG *r32,r/m32*	3/5			Exchange dword register with EA dword
90+ r	XCHG AX,*r16*	3	3	3	Exchange word register with AX
90+ r	XCHG *r16*,AX	3	3	3	Exchange word register with AX
90+ r	XCHG EAX,*r32*	3			Exchange dword register with EAX
90+ *r*	XCHG *r32*,EAX	3			Exchange dword register with EAX

XCHG exchanges two operands. The operands can be in either order. If a memory operand is involved, BUS LOCK is asserted for the duration of the exchange, regardless of the presence or absence of the LOCK prefix or of the value of the IOPL.

XLAT
XLATB

Table Look-up Translation

O D I T S Z A P C

Opcode	Instruction	Clocks			Description
		386	286	86	
D7	XLAT m8	5	5	11	Set AL to memory byte DS:[(E)BX + unsigned AL]
D7	XLATB	5	5	11	Set AL to memory byte DS:[(E)BX + unsigned AL]

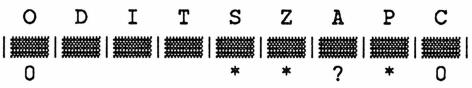
XLAT changes the AL register from the table index to the table entry. AL should be the unsigned index into a table addressed by DS:BX (for an address-size attribute of 16 bits) or DS:EBX (for an address-size attribute of 32 bits).

The operand to XLAT allows for the possibility of a segment override. XLAT uses the contents of BX even if they differ from the offset of the operand. The offset of the operand should have been moved into BX/EBX with a previous instruction.

The no-operand form, XLATB, can be used if the BX/EBX table will always reside in the DS segment.

XOR

Logical Exclusive OR



Opcode	Instruction	Clocks			Description
		386	286	86	
34 *ib*	XOR AL,*imm8*	2	3	4	Exclusive-OR immediate byte to AL
35 *iw*	XOR AX,*imm16*	2	3	4	Exclusive-OR immediate word to AX
35 *id*	XOR EAX,*imm32*	2			Exclusive-OR immediate dword to EAX
80 /6 *ib*	XOR *r/m8,imm8*	2/7	3/7	4/17+EA	Exclusive-OR immediate byte to *r/m* byte
81 /6 *iw*	XOR *r/m16,imm16*	2/7	3/7	4/17+EA	Exclusive-OR immediate word to *r/m* word
81 /6 *id*	XOR *r/m32,imm32*	2/7			Exclusive-OR immediate dword to *r/m* dword

83 /6 *ib*	XOR *r/m16,imm8*	2/7			XOR sign-extended immediate byte to *r/m* word
83 /6 *ib*	XOR *r/m32,imm8*	2/7			XOR sign-extended immediate byte to *r/m* dword
30 /*r*	XOR *r/m,r8*	2/6	2/7	3/16+EA	Exclusive-OR byte register to *r/m* byte
31 /*r*	XOR *r/m16,r16*	2/6	2/7	3/16+EA	Exclusive-OR word register into *r/m* word
31 /*r*	XOR *r/m32,r32*	2/6			Exclusive-OR dword register to *r/m* dword
32 /*r*	XOR *r8,r/m8*	2/7	2/7	3/9+EA	Exclusive-OR *r/m* byte to byte register
33 /*r*	XOR *r16,r/m16*	2/7	2/7	3/9+EA	Exclusive-OR *r/m* word to word register
33 /*r*	XOR *r32,r/m32*	2/7			Exclusive-OR to *r/m* dword to dword register

XOR computes the exclusive OR of the two operands. Each bit of the result is 1 if the corresponding bits of the operands are different; each bit is 0 if the corresponding bits are the same. The answer replaces the first operand.

P A R T 5

Coprocessor instructions

This section lists the 80x87 instructions in alphabetical order.

There is one entry for each combination of operand types that can be coded with the mnemonic. The following table explains the operand identifiers used in this section.

Identifier	Explanation
ST	Stack top; the register currently at the top of the stack.
ST(1)	A register in the stack $i(0 \le i \le 7)$ stack elements from the top. ST(1) is the next-on-stack register, ST(2) is below ST(1), etc.
Short-real	A short real (32 bits) number in memory.
Long-real	A long real (64 bits) number in memory.
Temp-real	A temporary real (80 bits) number in memory.
Packed-decimal	A packed decimal integer (18 digits, 10 bytes) in memory.
Word-integer	A word binary integer (16 bits) in memory.
Short-integer	A short binary integer (32 bits) in memory.
Long-integer	A long binary integer (64 bits) in memory.
nn-bytes	A memory area *nn* bytes long.

Here is a summary of the possible exceptions each instruction can cause:

- IS = invalid operand due to stack overflow/underflow
- I = invalid operand due to other cause
- D = denormal operand
- Z = zero-divide
- O = Overflow
- U = Underflow
- P = Inexact result (precision)

FABS

Absolute value

Exceptions: I

FABS (no operands)

Operands	Execution clocks			Code bytes	Example
	87	287	387		
No operands	10-17	10-17	22	2	FABS

FADD

Add real

Exceptions: I, D, O, U, P

FADD / /source/destination, source

Operands	Execution clocks			Code bytes	Example
	87	287	387		
//ST,ST(i)/ ST(i),ST	70-100	70-100	23-34	2	FADD ST,ST(4)
short real	90-120+EA	90-120	24-32	2-4	FADD AIR_TEMP[SI]
long real	95-125+EA	95-125	29-37	2-4	FADD [BX].MEAN

FADDP

Add real and pop

Exceptions: I, D, O, U, P

FADDP destination, source

Operands	Execution clocks			Code bytes	Example
	87	287	387		
ST(i),ST	75-105	75-105	23-34	2	FADDP ST(2),ST

FBLD

Packed decimal (BCD) load

Exceptions: I

FBLD source

Operands	Execution clocks			Code bytes	Example
	87	287	387		
Packed decimal	290-310	290-310	5	2-4	FBLD YTD_SALES

FBSTP

Packed decimal (BCD) store and pop

Exceptions: I

FBSTP destination

Operands	Execution clocks			Code bytes	Example
	87	287	387		
Packed decimal	520-540+EA	520-540+EA	512-534	2-4	FBSTP [BX].FORECAST

FCHS

Change sign

Exceptions: I

FCHS (no operands)

Operands	Execution clocks			Code bytes	Example
	87	287	387		
No operands	10-17	10-17	24-25	2	FCHS

FCLEX
FNCLEX

Clear exceptions

Exceptions: None

FCLEX/FNCLEX (no operands)

Operands	Execution clocks			Code bytes	Example
	87	287	387		
No operands	2-8	2-8	11	FNCLEX	

FCOM

Compare real

Exceptions: I, D

FCOM //source

Operands	Execution clocks			Code bytes	Example
	87	287	387		
//ST(i)	40-50	40-50	24	FCOM ST(1)	

short real	60-70+EA	60-70	26	2-4	FCOM [BP].UPPER_LIMIT
long real	65-75+EA	65-75	31	2-4	FCOM WAVELENGTH

FCOMP

Compare real and pop

Exceptions: I, D

FCOMP //source

Operands	Execution clocks			Code bytes	Example
	87	287	387		
//ST(i)	42-52	45-52	26	2	FCOMP ST(2)
short real	63-73+EA	63-73	26	2-4	FCOMP [BP+2].N_READINGS
long real	67-77+EA	67-77	31	2-4	FCOMP DENSITY

FCOMPP

Compare real and pop twice

Exceptions: I, D

FCOMPP (no operands)

Operands	Execution clocks			Code bytes	Example
	87	287	387		
No operands	45-55	45-55	26	2	FCOMPP

FCOS

Cosine of ST(0)

Exceptions: IS, I, D, U, P

FCOS

Operands	Execution clocks			Code bytes	Example
	87	287	387		
No operands			123-772*	2	FCOS

* These timings hold for operands in the range $|x| < 9/4$. For operands not in this range, up to 76 additional clocks may be needed to reduce the operand.

FDECSTP

Decrement stack pointer

Exceptions: None

FDECSTP (no operands)

Operands	Execution clocks			Code bytes	Example
	87	287	387		
No operands	6-12	6-12	22	2	FDECSTP

FDISI
FNDISI

Disable interrupts

8087 only

Exceptions: None

FDISI (no operands)

Operands	Execution clocks: Typical	Range	Operand word transfers	Code bytes	Coding example
No operands	5	2-8	0	2	FDISI

FDIV

Divide real

Exceptions: I, D, Z, O, U, P

FDIV //source/destination, source

Operands	Execution clocks			Code bytes	Example
	87	287	387		
//ST(i),ST	193-203	193-203	88-91	2	FDIV
short real	215-225	215-225	89	2-4	FDIV DISTANCE
long real	220-230	220-230	94	2-4	FDIV ARC[DI]

FDIVP

Divide real and pop

Exceptions: I, D, Z, O, U, P

FDIVP destination, source

Operands	Execution clocks			Code bytes	Example
	87	287	387		
//ST(i),ST	197-207	198-209	88-91	2	FDIVP ST(4),ST

FDIVR

Divide real reversed

Exceptions: I, D, Z, O, U, P

FDIVR //source/destination, source

Operands	Execution clocks			Code bytes	Example
	87	287	387		
//ST,ST(i)/ ST(i),ST	194-204	198-208	88-91	2	FDIVR ST(2),ST
short real	216-226+EA	215-225	89	2-4	FDIVR [BX].PULSE_RATE
long real	221-231+EA	220-230	94	2-4	FDIVR RECORDER.FREQUENC

FDIVRP

Divide real reversed and pop

Exceptions: I, D, Z, O, U, P

FDIVRP destination, source

Operands	Execution clocks			Code bytes	Example
	87	287	387		
ST(i),ST	198-208	198-208	88-91	2	FDIVRP ST(1),ST

FENI
FNENI

Enable interrupts

8087 only

Exceptions: None

FENI (no operands)

Operands	Execution clocks: Typical	Range	Operand word transfers	Code bytes	Coding example
(no operands)	5	2-8	0	2	FNENI

FFREE

Free register

Exceptions: None

FFREE destination

Operands	Execution clocks			Code bytes	Example
	87	287	387		
ST(i)	9-16	9-16	18	2	FFREE ST(1)

FIADD

Integer add

Exceptions: I, D, O, P

FIADD source

Operands	Execution clocks			Code bytes	Example
	87	287	387		
word integer	102-137+EA	102-137	71-85	2-4	FIADD DISTANCE_TRAVELLED
short integer	108-143+EA	108-143	57-72	2-4	FIADD PULSE_COUNT [SI]

FICOM

Integer compare

Exceptions: I, D

FICOM source

Operands	Execution clocks			Code bytes	Example
	87	287	387		
word integer	72-86+EA	72-86	71-75	2-4	FICOM TOOL.N_PASSES
short integer	78-91+EA	78-91	56-63	2-4	FICOM [BP+4].PARM_COUNT

FICOMP

Integer compare and pop

Exceptions: I, D

FICOMP source

Operands	Execution clocks			Code bytes	Example
	87	287	387		
word integer	74-88+EA	74-88	71-75	2-4	FICOMP [BP].LIMIT [SI]
short integer	80-93+EA	80-93	56-63	2-4	FICOMP N_SAMPLES

FIDIV

Integer divide

Exceptions: I, D, Z, O, U, P

FIDIV source

Operands	Execution clocks			Code bytes	Example
	87	287	387		
word integer	224-238+EA	224-238	136-140	2-4	FIDIV SURVEY.OBSERVATIONS
short integer	230-243+EA	230-243	120-127	2-4	FIDIV RELATIVE_ANGLE [DI]

FIDIVR

Integer divide reversed

Exceptions: I, D, Z, O, U, P

FIDIVR source

Operands	Execution clocks			Code bytes	Example
	87	287	387		
word integer	225-239+EA	224-238	135-141	2-4	FIDIVR [BP].X_COORD
short integer	231-245+EA	230-243	121-128	2-4	FIDIVR FREQUENCY

FILD

Integer load

Exceptions: I

FILD source

Operands	Execution clocks			Code bytes	Example
	87	287	387		
word integer	46-54+EA	46-54	61-65	2-4	FILD [BX].SEQUENCE
short integer	52-60+EA	52-60	45-52	2-4	FILD STANDOFF [DI]
long integer	60-68+EA	60-68	56-67	2-4	FILD RESPONSE.COUNT

FIMUL

Integer multiply

Exceptions: I, D, O, P

FIMUL source

Operands	Execution clocks			Code bytes	Example
	87	287	387		
word integer	124-138+EA	124-138	76-87	2-4	FIMUL BEARING
short integer	130-144+EA	130-144	61-82	2-4	FIMUL POSITION.Z_AXIS

FINCSTP

Increment stack pointer

Exceptions: None

FINCSTP (no operands)

Operands	Execution clocks			Code bytes	Example
	87	287	387		
No operands	6-12	6-12	21	2	FINCSTP

FINIT
FNINIT

Initialize processor

Exceptions: None

FINIT/FNINIT (no operands)

Operands	Execution clocks			Code bytes	Example
	87	287	387		
No operands	2-8	2-8	33	2	FINIT

FIST

Integer store

Exceptions: I, P

FIST destination

Operands	Execution clocks			Code bytes	Example
	87	287	387		
word integer	80-90+EA	80-90	82-95	2-4	FIST OBS.COUNT [SI]
short integer	82-92+EA	82-92	79-93	2-4	FIST [BP;].FACTORED_PULSE

FISTP

Integer store and pop

Exceptions: I, P

FISTP destination

Operands	Execution clocks			Code bytes	Example
	87	287	387		
word integer	82-92+EA	82-92	82-95	2-4	FISTP [BX]. ALPHA_COUNT [SI]
short integer	84-94+EA	84-94	79-93	2-4	FISTP CORRECTED_TIME
long integer	94-105+EA	94-105	80-97	2-4	FISTP PANEL. N_READINGS

FISUB

Integer subtract

Exceptions: I, D, O, P

FISUB source

Operands	Execution clocks			Code bytes	Example
	87	287	387		
word integer	102-137+EA	102-137	71-83	2-4	FISUB BASE_FREQUENCY
short integer	108-143+EA	108-143	57-82	2-4	FISUB TRAIN_SIZE [DI]

FISUBR

Integer subtract reversed

Exceptions: I, D, O, P

FISUBR source

Operands	Execution clocks			Code bytes	Example
	87	287	387		
word integer	103-139+EA	102-137	72-84	2-4	FISUBR FLOOR [BX][SI]
short integer	109-144+EA	108-143	58-83	2-4	FISUBR BALANCE

FLD

Load real

Exceptions: I, D

FLD source

Operands	Execution clocks			Code bytes	Example
	87	287	387		
ST(i)	17-22	17-22	14	2	FLD ST(0)
short real	38-56+EA	38-56	20	2-4	FLD READING [SI].PRESSURE
long real	40-60+EA	40-60	25	2-4	FLD [BP].TEMPERATURE
Temp real	53-65+EA	53-65	44	2-4	FLD SAVEREADING

FLDCW

Load control word

Exceptions: None

FLDCW source

Operands	Execution clocks			Code bytes	Example
	87	287	387		
2 bytes	7-14+EA	7-14	19	2-4	FLDCW CONTROL_WORD

FLDENV

Load environment

Exceptions: None

FLDENV source

Operands	Execution clocks			Code bytes	Example
	87	287	387		
14 bytes	35-45+EA	35-45	71	2-4	FLDENV [BP+6]

FLDLG2

Load $\log_{10}2$

Exceptions: I

FLDLG2 (no operands)

Operands	Execution clocks			Code bytes	Example
	87	287	387		
No operands	18-24	18-24	41	2	FLDLG2

FLDLN2

Load $\log_e 2$

Exceptions: I

FLDLN2 (no operands)

Operands	Execution clocks			Code bytes	Example
	87	287	387		
No operands	17-23	17-23	41	2	FLDLN2

FLDL2E

Load $\log_2 e$

Exceptions: I

FLDL2E (no operands)

Operands	Execution clocks			Code bytes	Example
	87	287	387		
No operands	15-21	15-21	40	2	FLDL2E

FLDL2T

Load $\log_2 10$

Exceptions: I

FLDL2T (no operands)

Operands	Execution clocks			Code bytes	Example
	87	287	387		
No operands	16-22	16-22	40	2	FLDL2T

FLDPI

Load Π (pi)

Exceptions: I

FLDPI (no operands)

Operands	Execution clocks			Code bytes	Example
	87	287	387		
No operands	16-22	16-22	40	2	FLDPI

FLDZ

Load +0.0

Exceptions: I

FLDZ (no operands)

Operands	Execution clocks			Code bytes	Example
	87	287	387		
No operands	11-17	11-17	20	2	FLDZ

FLD1

Load +1.0

Exceptions: I

FLD1 (no operands)

Operands	Execution clocks			Code bytes	Example
	87	287	387		
No operands	15-21	15-21	24	2	FLD1

FMUL

Multiply real

Exceptions: I, D, O, U, P

FMUL //source/destination,source

Operands	Execution clocks			Code bytes	Example
	87	287	387		
//ST(i),ST/ST, ST(1)*	90-105	90-145	29-57	2	FMUL ST,ST(3)
//ST(i),ST/ST ST,ST(1)	130-145	90-145	29-57	2	FMUL ST,ST(3)
short real	110-125+EA	110-125	27-35	2-4	FMUL SPEED_FACTOR
long real*	112-126+EA	112-168	32-57	2-4	FMUL [BP].HEIGHT
long real	154-168+EA	112-168	32-57	2-4	FMUL [BP].HEIGHT

* Occurs when one or both operands is "short"—it has 40 trailing zeros in its fraction (for example, it was loaded from a short-real memory operand).

FMULP

Multiply real and pop

Exceptions: I, D, O, U, P

FMULP destination,source

Operands	Execution clocks			Code bytes	Example
	87	287	387		
ST(i),ST*	94-108	198-208	29-57	2	FMULP ST(1),ST
ST(i),ST	134-148	198-208	29-57	2	FMULP ST(1),ST

* Occurs when one or both operands is "short"—it has 40 trailing zeros in its fraction (for example, it was loaded from a short-real memory operand).

FNOP

No operation

Exceptions: None

FNOP (no operands)

Operands	Execution clocks			Code bytes	Example
	87	287	387		
No operands	10-16	10-16	12	2	FNOP

FPATAN

Partial arctangent

Exceptions: U, P (operands not checked)

FPATAN (no operands)

Operands	Execution clocks			Code bytes	Example
	87	287	387		
No operands	250-800	250-800	314-487	2	FPATAN

FPREM

Partial remainder

Exceptions: I, D, U

FPREM (no operands)

Operands	Execution clocks			Code bytes	Example
	87	287	387		
No operands	15-190	15-190	74-155	2	FPREM

FPTAN

Partial tangent

Exceptions: I, P (operands not checked)

FPTAN (no operands)

Operands	Execution clocks			Code bytes	Example
	87	287	387		
No operands	30-540	30-540	191-573	2	FPTAN

FRNDINT

Round to integer

Exceptions: I, P

FRNDINT (no operands)

Operands	Execution clocks			Code bytes	Example
	87	287	387		
No operands	16-50	16-50	66-80	2	FRNDINT

FRSTOR

Restore saved state

Exceptions: None

FRSTOR source

Operands	Execution clocks			Code bytes	Example
	87	287	387		
94 bytes	197-207+EA	205-215	308	2-4	FRSTOR [BP]

* The 80287 execution clock count for this instruction is not meaningful in determining overall instruction execution time. For typical frequency ratios of the 80286 and 80287 clocks, 80287 execution occurs in parallel with the operand transfers. The operand transfers determine the overall execution time of the instructions. For 80286:80287 clock frequency ratios of 4:8, 1:1, and 8:5, the overall execution clock count for this instruction is estimated at 490, 302, and 227 80287 clocks, respectively.

FSAVE
FNSAVE

Save state

Exceptions: None

FSAVE/FNSAVE destination

Operands	Execution clocks			Code bytes	Example
	87	287	387		
94 bytes	197-207+EA	205-215	375-376	2-4	FSAVE [BP]

* The 80287 execution clock count for this instruction is not meaningful in determining overall instruction execution time. For typical frequency ratios of the 80286 and 80287 clocks, 80287 execution occurs in parallel with the operand transfers. The operand transfers determine the overall execution time of the instruction. For 80286:80287 clock frequency ratios of 4:8, 1:1, and 8:5, the overall execution clock count for this instruction is estimated at 376, 233, and 174 80287 clocks, respectively.

FSCALE

Scale

Exceptions: I, O, U

FSCALE (no operands)

Operands	Execution clocks			Code bytes	Example
	87	287	387		
No operands	32-38	32-38	67-86	2	FSCALE

FSETPM

Set protected mode

Exceptions: None

FSETPM (no operands)

Operands	Execution clocks: Typical	Range	Operand word transfers	Code bytes	Coding example
No operands		2-8	0	2	FSETPM

FSIN

Sine of ST(0)

Exceptions: IS, I, D, U, P

FSIN

Operands	Execution clocks: Typical	Range	Operand word transfers	Code bytes	Coding example
No operands			122-771*	2	FSIN

* These timings hold for operands in the range $|x| < 3/4$. For operands not in this range, up to 76 additional clocks may be needed to reduce the operand.

FSINCOS

Sine and cosine of ST(0)

Exceptions: IS, I, D, U, P

FSINCOS

Operands	Execution clocks: Typical	Range	Operand word transfers	Code bytes	Coding example
No operands			194-809*	2	FSINCOS

* These timings hold for operands in the range $|x| < \pi/4$. For operands not in this range, up to 76 additional clocks may be needed to reduce the operand.

FSQRT

Square root

Exceptions: I, D, P

FSQRT (no operands)

Operands	Execution clocks			Code bytes	Example
	87	287	387		
No operands	180-186	180-186	122-129	2	FSQRT

FST destination

Store real

Exceptions: I, O, U, P

FST destination

Operands	Execution clocks			Code bytes	Example
	87	287	387		
ST(i)	15-22	15-22	11	2	FST ST(3)
short real	84-90+EA	84-90	44	2-4	FST CORRELATION [DI]
long real	96-104+EA	96-104	45	2-4	FST MEAN_READING

FSTCW
FNSTCW

Store control word

Exceptions: None

FSTCW destination

Operands	Execution clocks			Code bytes	Example
	87	287	387		
2 bytes	12-18+EA	12-18	15	2-4	FSTCW SAVE_CONTROL

FSTENV
FNSTENV

Store environment

Exceptions: None

FSTENV destination

Operands	Execution clocks			Code bytes	Example
	87	287	387		
14 bytes	40-50+EA	40-50	103-104	2-4	FSTENV [BP]

FSTP

Store real and pop

Exceptions: I, O, U, P

FSTP destination

Operands	Execution clocks			Code bytes	Example
	87	287	387		
ST(i)	17-24	17-24	12	2	FSTP ST(2)
short real	86-92+EA	86-92	44	2-4	FSTP [BX]. ADJUSTED_RPM
long real	98-106+EA	98-106	45	2-4	FSTP TOTAL_DOSAGE
Temp real	52-58+EA	52-58	53	2-4	FSTP REG_SAVE [SI]

FSTSW
FNSTSW

Store status word

Exceptions: None

FSTSW/FNSTSW destination

Operands	Execution clocks			Code bytes	Example
	87	287	387		
2 bytes	12-18+EA	12-18	15	2-4	FSTSW SAVE_STATUS

FSTSW AX
FNSTSW AX

Store status word to AX

Exceptions: None

FSTSW destination

Operands	Execution clocks			Code bytes	Example
	87	287	387		
AX		10-16	13	2	FSTSW AX

FSUB

Subtract real

Exceptions: I, D, O, U, P

FSUB //source/destination,source

Operands	Execution clocks			Code bytes	Example
	87	287	387		
//ST,ST/(i)/ ST(i),ST	70-100	70-100	26-37	2	FSUB ST,ST(2)
short real	90-120+EA	90-120	24-32	2-4	FSUB BASE_VALUE
long real	95-125+EA	95-125	28-36	2-4	FSUB COORDINATE.X

FSUBP

Subtract real and pop

Exceptions: I, D, O, U, P

FSUBP destination, source

Operands	Execution clocks			Code bytes	Example
	87	287	387		
ST(i),ST	75-105	75-105	26-37	2	FSUBP ST(2),ST

FSUBR

Subtract real reversed

Exceptions: I, D, O, U, P

FSUBR //source/destination, source

Operands	Execution clocks			Code bytes	Example
	87	287	387		
//ST,ST(i)/ ST(i),ST	70-100	70-100	26-37	2	FSUBR ST,ST(1)
short real	90-120+EA	90-120	25-33	2-4	FSUBR VECTOR [SI]
long real	95-125+EA	95-125	29-37	2-4	FSUBR [BX].INDEX

FSUBRP

Subtract real reversed and pop

Exceptions: I, D, O, U, P

FSUBRP destination, source

Operands	Execution clocks			Code bytes	Example
	87	287	387		
ST(i),ST	75-105	75-105	26-37	2	FSUBRP ST(1),ST

FTST

Test stack top against +0.0

Exceptions: I, D

FTST (no operands)

Operands	Execution clocks			Code bytes	Example
	87	287	387		
No operands	38-48	38-48	28	2	FTST

FUCOM

Unordered compare

Exceptions: IS, I, D

Operands	Execution clocks: Typical	Range	Operand word transfers	Code bytes	Coding example
//ST(i)			24	2	FUCOM ST(1)

FUCOMP

Unordered compare

Exceptions: IS, I, D

Operands	Execution clocks: Typical	Range	Operand word transfers	Code bytes	Coding example
//ST(i)			26	2	FUCOMP ST(2)

FUCOMPP

Unordered compare

Exceptions: IS, I, D

Operands	Execution clocks: Typical	Range	Operand word transfers	Code bytes	Coding example
No operands			26	2	FUCOMPP

FWAIT

Wait while 80287 is busy

Exceptions: None (CPU instruction)

FWAIT (no operands)

Operands	Execution clocks: Typical	Range	Operand word transfers	Code bytes	Coding example
No operands	3+5*n**	3+5*n*	0	1	FWAIT

n = number of time CPU examines BUSY line before 80287 completes execution of previous instruction.

FXAM

Examine stack top

Exceptions: None

FXAM (no operands)

Operands	Execution clocks			Code bytes	Example
	87	287	387		
No operands	12-23	12-23	30-38	2	FXAM

FXCH

Exchange registers

Exceptions: I

FXCH //destination

Operands	Execution clocks			Code bytes	Example
	87	287	387		
//ST(i)	10-15	10-15	18	2	FXCH ST(2)

FXTRACT

Extract exponent and significant

Exceptions: I

FXTRACT (no operands)

Operands	Execution clocks			Code bytes	Example
	87	287	387		
No operands	27-55	27-55	70-76	2	FXTRACT

FYL2X

$Y * \log_2 X$

Exceptions: P (operands not checked)

FYL2X (no operands)

Operands	Execution clocks			Code bytes	Example
	87	287	387		
No operands	900-1100	900-1100	120-538	2	FYL2X

FYL2XP1

$Y * \log_2(X+1)$

Exceptions: P (operands not checked)

FYL2XP1 (no operands)

Operands	Execution clocks			Code bytes	Example
	87	287	387		
No operands	700-1000	700-1000	257-547	2	FYL2XP1

F2XM1

2^x-1

Exceptions: U, P (operands not checked)

F2XM1 (no operands)

Operands	Execution clocks			Code bytes	Example
	87	287	387		
No operands	310-630	310-630	211-476	2	F2XM1

2.0

QUICK REFERENCE GUIDE

TURBO ASSEMBLER®

BORLAND

1800 GREEN HILLS ROAD, P.O. BOX 660001, SCOTTS VALLEY, CA 95067-0001, (408) 438-5300
UNIT 8 PAVILIONS, RUSCOMBE BUSINESS PARK, TWYFORD, BERKSHIRE, RG10 9NN-ENGLAND
43 AVENUE DE L'EUROPE—BP 6, 78141 VELIZY VILLACOUBLAY CEDEX, FRANCE
PART # 15MN-ASD04-20 ■ BOR 1482